Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Définition d'un langage de consultation pour une Base de Connaissances "orientée-objet"

Nachtergaele, Veronique

Award date:
1989

Awarding institution:
Universite de Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 05. May. 2023

Définition d'un langage de consultation pour une Base de Connaissances "orientée-objet"

Véronique Nachtergaele.

Mémoire réalisé sous la
direction du Professeur B. Le Charlier
en vue de l'obtention du titre de
Licencié et Maître en Informatique.

Namur Juin 1989.

Bref résumé du mémoire

Ce mémoire consiste à définir et à implémenter PCL, un langage de consultation de Bases de Connaissances orientées-objet, qui constitue l'un des composants d'un environnement de prototypage développé par la société BIM dans le cadre du projet ESPRIT 892 DAIDA. Il étudie d'abord les liens entre l'approche orientée-objet et la programmation logique, qui sont les deux domaines principaux auxquels touche le langage PCL. Il présente ensuite un certain nombre de théories et méthodes permettant de définir un langage formalisé. Il présente finalement l'implémentation de PCL qui a été réalisée en PROLOG.

Mots-clés: approche orientée-objet, programmation logique, bases de connaissances, langages formels, syntaxe abstraite, syntaxe concrète, sémantique, grammaires attribuées, analyse syntaxique.

Abstract

This thesis presents the definition and implementation of PCL, an object-oriented knowledge bases consultation language. This language is part of a prototype-based software engineering environment, currently developped at BIM, under the DAIDA 892 ESPRIT project. This work investigates the relations between the object-oriented and logic programming approaches, to which PCL is highly related. It discusses the theoretical and methodological aspects of formal language definition. It presents the PROLOG implementation of the PCL language.

Keywords: object-oriented approach, logic programming, knowledge bases, formal languages, abstract syntax, concrete syntax, semantics, attributed grammars, parsing.

Remerciements

Je remercie d'abord la société BIM, qui m'a donné l'occasion d'effectuer un stage dans le cadre ce mémoire, et tout particulièrement les membres du Research and Development Team, pour la gentillesse de leur accueil.

Je remercie ensuite Jean-Marc Trinon, qui m'a assistée tout au long de mon stage, pour ses conseils amicaux, sa disponibilité et son aide précieuse.

Je remercie également Marc Derroitte, pour sa lecture attentive du manuscrit, ainsi que ses remarques constructives.

Je remercie enfin Baudouin Le Charlier, promoteur de ce mémoire, pour l'intérêt du sujet qu'il m'a proposé, ainsi que pour le temps qu'il a bien voulu me consacrer lors de l'élaboration de la partie théorique de ce travail.

Table des matières

Introduction 1

Première partie : Position du problème 3

1. Le projet DAIDA 4
2. Objectifs du projet 4
3. Méthodologie proposée 5
   - 3.1. Elaboration du modèle conceptuel 5
   - 3.2. Validation du modèle par prototypage 5
   - 3.3. Derivation du software 6
4. Outils 6
5. Architecture 7
   - 5.1. Le composant GKBMS 7
   - 5.2. Description de l'architecture 8
6. Environnement de prototypage 9
   - 6.1. Description générale de l'environnement 10
   - 6.2. TDL: Taxis Design Language 12
   - 6.3. PROBE: Prolog Object Extension 14
   - 6.4. PCL: Prolog Constraint Language 18
7. Objectifs du mémoire 19

Deuxième partie : Cadre informatique du mémoire 20

1. Introduction 21
2. L'approche orientée-objet 22
   - 2.1. Apparition de l'approche orientée-objet 22
   - 2.2. Caractéristiques générales de l'approche orientée-objet 23
3. Approche orientée-objet et programmation logique 27
   - 3.1. L'extension du langage de programmation logique 27
   - 3.2. L'ajout d'un "package" orienté-objet au langage de programmation logique 28
   - 3.3. L'utilisation de concepts orientés-objet dans le langage de programmation logique 28
   - 3.4. Utilisation de PROLOG comme outil d'implémentation 28

Troisième partie : Théories et méthodes 29

Chapitre 1: Méta-modèle de Bases de Connaissances

1. Le contexte de définition d'un langage 32
2. Cas des langages de requêtes 33
3. Méthode de définition 34
   3.1. Une methode possible 34
   3.2. La méthode proposée 34

Chapitre 2: Définition d'un langage formalisé

1. Introduction 37
2. Présentation de la démarche générale 37
   2.1. Identification des concepts du langage 37
   2.2. Syntaxe et sémantique 37
   2.3. Syntaxe abstraite et syntaxe concrete 38
   2.4. Récapitulation 40
3. Identification des concepts du langage 40
4. Syntaxe abstraite et grammaires attribuées 41
   4.1. Definition de la syntaxe abstraite 41
   4.2. Correction syntaxique d'une expression abstraite 42
5. Sémantique 47
   5.1. Intérêt de la définition de la sémantique 47
   5.2. Méthodes de définition de la sémantique 48
6. Syntaxe concrète 51

Chapitre 3: Implémentation d'un langage

1. Introduction 53
2. Remarques préliminaires 53
3. Présentation des éléments du programme 54
4. Représentation interne des expressions 55
   4.1. Principe general 55
   4.2. Représentation basée sur l'expression abstraite 55
   4.3. Représentation basée sur l'expression concrète 56
5. Le composant syntaxique 57
   5.1. Hypothese de travail 57
   5.2. Phases de l'analyse syntaxique 57
   5.3. Techniques d'analyse syntaxique utilisables en PROLOG 59
6. Le composant sémantique 70

Quatrième partie : Le langage PCL 71

Chapitre 1: Méta-modèle de Bases de Connaissances

1. Introduction 74
2. Presentation de la hierarchie 74
3. Niveau de cohérence ZERO 75
4. Niveau de cohérence UN 75
5. Niveau de cohérence DEUX 76
6. Niveau de cohérence TROIS 76
7. Niveau de cohérence QUATRE 76

Chapitre 2: Définition du langage PCL

1. Introduction 78
2. Identification des concepts de PCL 78
3. Syntaxe abstraite de PCL 79
   3.1. Définition de la syntaxe abstraite 79
   3.2. Règles de correction syntaxique 80
4. Sémantique de PCL 83
   4.1. L'attribut "eval" 83
   4.2. L'attribut "denote" 83
   4.3. L'attribut "insForEval" 84
   4.4. Les attributs "inh-Table" et "Table" 84
5. Syntaxe concrète de PCL 85
   5.1. Grammaire BNF 85
   5.2. Conditions syntaxiques supplémentaires 86

Chapitre 3: Implémentation du langage PCL

1. Introduction 88
2. Representation interne des expressions 88
3. Le composant syntaxique 88
   3.1. Conversion de la grammaire BNF 89
   3.2. Définition des opérateurs PROLOG 90
   3.3. Vérification des conditions supplémentaires 90
4. Le composant sémantique 91
5. Le traitement des erreurs 91

Conclusion 92

Références bibliographiques 94

Introduction

Ce mémoire consiste à définir et à implémenter le langage PCL, qui constitue l'un des composants d'un environnement de prototypage, développé par la société BIM, dans le cadre du projet ESPRIT 892 DAIDA. Il est divisé en quatre parties.

La première, "Position du problème", présente le projet DAIDA, et y situe le langage PCL en déterminant les objectifs du travail.

La deuxième, "Cadre informatique", étudie les liens entre l'approche orientée-objet et la programmation logique, qui sont les deux domaines principaux auxquels touche le langage PCL.

La troisième partie "Théories et méthodes", présente les bases théoriques qui ont permis de définir ce langage.

Finalement, la quatrième partie, "Le langage PCL", montre comment ces théories ont été mises en oeuvre pour atteindre les objectifs du mémoire. Elle constitue une présentation succincte de la démarche suivie pour définir et implémenter le langage PCL. Cette démarche est reprise complètement et en détail dans les annexes I à II.

Première partie

Position du problème

1. Le projet DAIDA

Ce mémoire est une contribution au développement d'un environnement de prototypage s'inscrivant dans le cadre du projet ESPRIT 892 DAIDA (Development of Advanced Interactive Data-Intensive Applications).

Ce projet, qui a débuté en 1986 pour une durée de quatre ans, a pour objectif de réaliser un environnement de développement de systèmes d'informations orientés Base de Connaissances.

DAIDA est le fruit de la collaboration entre des institutions de recherche et des groupes industriels[1], ce qui permet de combiner des aspects pratiques et théoriques de production de software.

Cette collaboration a permis de rendre operationnels des prototypes de la plupart des composants de l'environnement de DAIDA, avec un interface de qualité acceptable pour l'utilisateur.

2. Objectifs du projet

Ce projet a pour but la création de langages, de méthodes et d'outils offrant un environnement orienté "Base de Connaissances".

Cet environnement doit permettre d'accroître la qualité et la productivité du développement et de la maintenance des systèmes softwares (plus particulièrement des systèmes d'informations utilisant des données de manière intensive).

Ses fondements théoriques consistent en:

- une approche orientée-objet (pour la représentation des connaissances),
- des méthodes de prototypage rapide basées sur la logique,
- des méthodes de production de software.

Selon les auteurs du projet ([JARKE]), beaucoup de recherches sont effectuées dans les domaines du développement de software et de la conception de Bases de Données, mais peu de recherches

---

[1] Ces partenaires sont :
BIM (Everberg / Belgium),
BP Research Center (Sunbury / UK),
GFI (Paris / France),
Johann Wolfgang Goethe-Universität. (Frankfurt / FRG),
Research Center of Crete (Iraklion / Greece),
SCS (Hamburg / FRG).
Universität Passau. (Passau / FRG).

tendent à combiner ces deux domaines. C'est dans cette voie que se situe le projet DAIDA.

3. Méthodologie proposée

Le projet DAIDA propose une méthodologie de production de software qui comprend les trois phases suivantes:

- Elaboration d'un modèle conceptuel du systeme,
- Validation de ce modèle par prototypage,
- Dérivation d'un software de haute qualité par la méthode des raffinements successifs.

Les points suivants décrivent ces différentes phases.

3.1. Elaboration du modèle conceptuel

Dans l'optique du projet DAIDA, l'élaboration d'un tel modèle prend en compte les paradigmes suivants:

- La spécification d'un système d'informations peut être supportee par une representation de Connaissances.

  Cette représentation combine des objets structurés (selon des hiérarchies de classification, d'agrégation ou de généralisation) avec des prédicats (assertions):

  - Les objets modélisent les éléments du monde, et définissent ainsi les concepts de base du monde réel.
  - Les prédicats permettent d'énoncer des règles et contraintes sur ces objets.

- Les systemes d'informations existent dans un monde en évolution.

  De ce fait, la représentation doit intégrer la notion de temps pour indiquer des intervalles valides de Connaissance au sujet des spécifications et des systèmes.

Pour satisfaire ces contraintes, deux langages orientés-objet (très fortement interconnectés) ont été développés. Il s'agit des langages SML/CML et TDL qui seront présentés dans la suite.

3.2. Validation du modèle par prototypage

Cette phase permet de valider le modèle conceptuel par une démonstration à l'utilisateur d'un prototype, de taille réduite et peut-être inefficace, mais pleinement fonctionnel.

Cet environnement de prototypage sera décrit au point **6.**, et comprend, entre autres composants, le langage PCL dont la définition et l'implémentation font l'objet de ce mémoire.

3.3. Dérivation du software

Après validation du modèle conceptuel, le système permet au concepteur de dériver un software opérationnel.

Cette dérivation est basée sur les fondements théoriques des langages de programmation de Bases de Données (Théorie relationnelle), à partir de spécifications orientées-objet.

Elle utilise des outils permettant de construire des spécifications correctes et prouvables de programmes modulaires dans le langage de programmation de Base de Données (DBPL), décrit ci-dessous.

4. Outils

Dans le cadre du projet DAIDA, on peut discerner différents niveaux permettant de décrire le développement d'un système d'informations, depuis les spécifications jusqu'à l'implémentation. Ces niveaux regroupent les outils qui mettent en oeuvre la méthodologie présentée au point précédent (Figure 1).

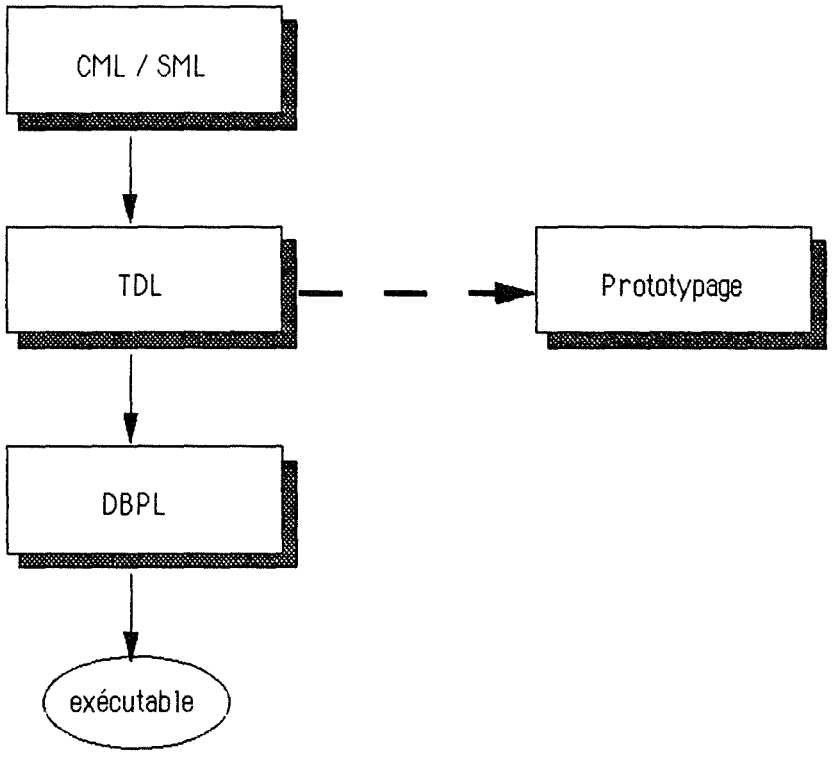

Figure 1 : Outils de l'environnement DAIDA

Ces niveaux se définissent comme suit:

- Le premier niveau consiste en la description des spécifications du système dans le langage SML/CML, qui constitue un langage de représentation de connaissances orienté-objet, permettant d'exprimer des spécifications sous la forme d'un modèle temporel du monde.

- Au second niveau, le langage TDL fournit une description de toutes les informations et les fonctionnalités du système, mais sans notion de temps.

  Cette description satisfait les spécifications données au premier niveau, et permet de laisser la prise de toute décision d'implémentation au troisième niveau (niveau d'implémentation).

  C'est à partir de cette description que l'on peut générer un prototype du système. Ceci est réalisé par la conversion du modèle TDL en une extension orientée-objet de BIM-Prolog.

- Le dernier niveau constitue l'implémentation du modèle conceptuel décrit par TDL dans le langage de programmation de Bases de Données DBPL. Cette dernière étape fournit un programme executable.

5. Architecture

Les différents outils décrits au point précédent sont regroupés en un systeme de développement de software, dont l'architecture est décrite ci-dessous.

5.1. Le composant GKBMS

Les relations entre les différents langages, ainsi que la gestion d'une Base de Connaissances pour suivre l'évolution des outils, méthodes et documents, sont un point crucial du projet. Ce contrôle de l'évolution du système est réalise par un composant appelé GKBMS.

Ce composant contient un modèle conceptuel du reste de l'environnement DAIDA, qui regroupe des informations aussi bien au niveau général (constructions du langage, outils disponibles dans l'environnement DAIDA) qu'au niveau des projets spécifiques aux applications (objets modélisés, transformations exécutées, outils utilisés pour un système d'informations particulier).

Plus particulièrement, la partie relative à l'objet modélisé (Design Object Knowledge) contient une représentation du Monde / Modèle du système, le modèle conceptuel de ce système, les programmes DBPL, ainsi que les représentations intermédiaires nécessaires entre les différentes etapes.

La partie relative au processus de modélisation (Design Process Knowledge) est formalisée par les catégories de tâches conceptuelles appelees classes de décisions, et par leurs instanciations dans un projet de software particulier.

La partie relative aux outils de modélisation (Design Tool Knowledge) permet aux développeurs de connaître les informations concernant les outils et documents utilisés, ainsi que leurs applications dans les décisions du modèle d'exécution.

5.2. Description de l'architecture

L'architecture de DAIDA peut être synthétisée par le schéma de la Figure 2.

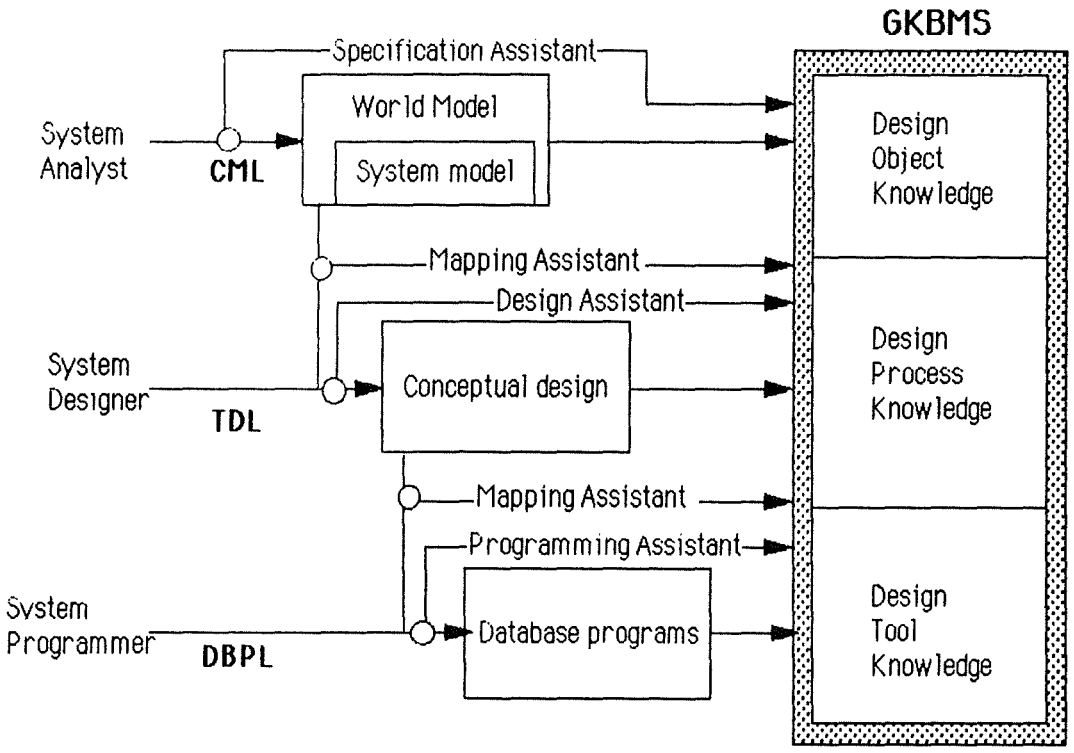

Figure 2 : Architecture de DAIDA

Cette figure montre les interactions entre les composants du système, lors de la transformation entre:

- un modèle SML/CML et son correspondant TDL,
- un modèle TDL et son correspondant DBPL.

Les points suivants détaillent ces transformations.

5.2.1. Transformation SML/CML - TDL

La transformation du modèle SML/CML en un modèle conceptuel satisfaisant les critères du langage TDL doit permettre de transformer un modèle temporel (dit historique) en un système a états de transition.

Cette transformation est facilitée par un outil constitué d'un ensemble de règles, chaque règle permettant de transformer une construction du langage SML/CML en une construction TDL (mapping assistant).

La tâche principale de cette transformation est de déterminer quelles informations doivent être représentées dans le système, comment elles interagissent, et quelles sont les relations temporelles qui doivent être transformées en données explicites, ou en transactions[1].

5.2.2. Transformation TDL - DBPL

La transformation d'un modèle conceptuel TDL en programmes DBPL se base sur une approche en trois étapes:

1. conversion des spécifications du modèle TDL en spécifications relationnelles, en réorganisant la hiérarchie orientée-objet en une structure mettant en oeuvre les principes de modularité et l'utilisation de types abstraits.

2. application des transformations de la theorie relationnelle, de manière à obtenir un modèle optimisé.

3. génération de programmes en langage DBPL à partir des spécifications finales obtenues à l'étape 2.

De manière similaire à la transformation du modèle SML/CML en TDL, un ensemble de définitions de modèles de transformations "faisables" facilite cette transformation (mapping assistant).

6. Environnement de prototypage

Après avoir décrit la méthodologie proposée par DAIDA, les outils supportant cette méthodologie, et l'architecture du système intègrant ces outils, on va détailler l'environnement de prototypage du système, dans lequel s'inscrit ce mémoire.

---

[1] Intuitivement, une transaction est une suite d'opérations sur une Base de Connaissances, qui doivent être exécutées complètement, ou pas du tout.

6.1. Description générale de l'environnement

L'environnement de prototypage utilise des langages de haut niveau permettant de combiner la puissance d'expression de la logique avec les caracteristiques de la representation orientee-objet.

6.1.1. Composants de l'environnement

L'environnement est composé des éléments suivants:

- Le langage **TDL**, servant à décrire le modèle conceptuel du système d'informations, [MEIR-TR-VEN] et [MEIR-TR].
- Un **interface "utilisateur"**, basé sur la notion de fenêtrage, [MEIR-TR]:

  il comprend un éditeur/traceur, un compilateur/debugger on-line, un tableur, ...
- Le langage **PROBE**, permettant l'accès à la Base de Connaissances, et réalisant la vérification de la cohérence de la Base de Connaissances, [MEIR-TR-VEN] et [MEIR-TR]:

  il doit réduire la distance conceptuelle entre TDL et PROLOG, en offrant des primitives orientées-objet implémentées en PROLOG,
- Le langage **PCL**, dont la définition et l'implémentation font l'objet de ce mémoire, [MEIR-TR]:

  il est constitué d'un langage de requêtes et d'expressions de contraintes, permettant de réduire la distance entre l'expression d'assertions TDL et des prédicats de type PROLOG.

6.1.2. Interaction entre les composants

La Figure 3 illustre les interactions entre les composants de l'environnement de prototypage.

Ces interactions peuvent être détaillées comme suit:

- TDL utilise les services de PROBE:

  Lorsque l'utilisateur consulte le modèle conceptuel TDL, il accède à la Base de Connaissances via le langage PROBE, pour obtenir des informations sur les éléments du modèle conceptuel ou sur des instances de ces éléments[1].

---

[1] En fait, le modèle conceptuel présent à ce niveau n'est pas exactement le modèle TDL, mais une extension de celui-ci, adaptée aux caractéristiques propres de l'environnement de prototypage.

- TDL utilise les services de PCL:

  Les expressions de contraintes sur les éléments du modèle conceptuel au niveau de TDL sont decrites dans le langage PCL.

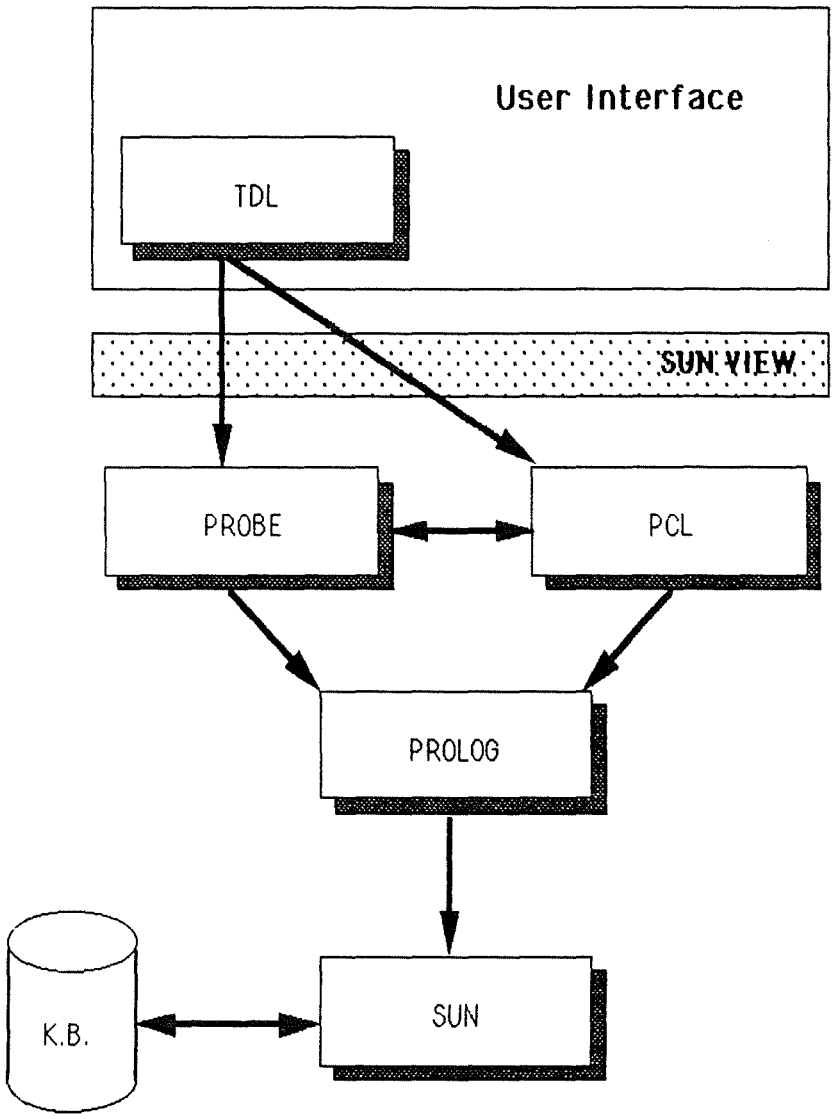

Figure 3 : Environnement de prototypage.

- PROBE utilise les services de PCL:

  Certaines requêtes d'accès à la Base de Connaissances peuvent contenir des (sous-)expressions décrites dans le langage PCL.

- PCL utilise les services de PROBE:

  L'analyse d'expressions PCL peut necessiter l'accès à certaines informations de la Base de Connaissances.

- PROBE et PCL utilisent PROLOG:

Les langage PROBE et PCL sont implémentés en BIM-PROLOG et sont donc utilisables sur SUN.

- La Base de Connaissances est une Base de Connaissances PROLOG.

  Tout accès doit donc se faire en PROLOG.

Après cette description générale de l'environnement de prototypage, les principaux composants vont être présentés plus en détail. Ces composants sont les langages TDL, PROBE, et PCL.

6.2. TDL: Taxis Design Language

6.2.1. Présentation de TDL

Rappelons que le langage TDL a pour but de décrire le modèle conceptuel du système d'informations.

Comme il a été précisé précédemment, le modèle conceptuel fourni au niveau de TDL est une description des informations et fonctionnalités du système d'informations, exempte de toute notion de temps.

Ceci induit les conséquences suivantes:

- Les données sont des entités groupées en classes.
- Les informations concernant les entités sont maintenues par des mises-à-jour destructives:

  Un état de la Base de Données décrit la connaissance courante du système d'informations, et toute mise-à-jour conduit à la création d'un nouvel état, sans mémorisation de l'ancien.

- Les transitions entre ces états de la Base de Connaissances sont representees par des transactions:

  Une transaction, grâce a un mecanisme similaire à celui des transactions sur une Base de Données, permet uniquement des transitions d'un état cohérent à un (autre) état cohérent.

  Elle autorise tant la consultation que la modification d'une Base de Connaissances.

  Elle est décrite dans le langage TDL, de manière similaire à la description d'une classe du modèle conceptuel. Elle spécifie des assertions décrivant les valeurs des paramètres et variables locales au début et à la fin de son exécution. Elle joue donc le rôle de "spécification exécutable".

6.2.2. Concepts fondamentaux

Un modèle conceptuel dans le langage TDL décrit le monde comme étant constitué d'entités reliées, et groupées en différents ensembles. Chaque ensemble est appele **classe**, et une entite d'un tel ensemble est une **instance** de la classe.

Les différents éléments du langage TDL sont les suivants:

- Les **classes** de données sont les objets de base du modèle conceptuel, et représentent des concepts génériques.
- Différentes classes ayant les mêmes caractéristiques sont groupées en une seule classe, appelée **métaclasse**. On distingue les métaclasses:

  Basic (classes integer, real, string, boolean),

  Enumerated (classes décrites par une liste finie de noms),

  Entity (classes représentant des ensembles d'entités de "nature" non précisée),

  Aggregate (produits cartésiens de valeurs).

- Une classe peut être vue comme une description (définition) des caracteristiques (ou attributs) de ses instances; chaque attribut est appelé **slot**.
- Un slot est défini par un ensemble de **facettes**.

  Les facettes permettent de préciser certaines propriétés du slot, comme la presence obligatoire ou optionnelle de sa valeur, le nombre de valeurs qu'il peut prendre, l'ensemble des valeurs permises, ...

  On distingue les slots dits de contrainte, exprimant une condition portant sur les autres slots de la classe, et les slots dits de propriété, dont la valeur est une référence (ou un ensemble de références) à une autre classe.

  Les différentes facettes sont:

  - category (indiquant s'il s'agit d'un slot de propriété ou de contrainte),
  - cardinality (nombres minimum et maximum de valeurs du slot),
  - inverse (définit un couple de propriétés inverses entre les classes),
  - default (valeur par défaut d'un slot),
  - comment (commentaire associé au slot).

La description TDL du modèle conceptuel ne prend en compte que les classes, leurs caractéristiques, et la manière dont elles sont reliees ou groupees. Les classes peuvent être organisees en hierarchies suivant la relation IS-A (principe de généralisation / spécialisation introduisant le concept d'héritage).

Un ensemble d'instances de classes d'un modèle conceptuel détermine constitue une instance de ce modèle.

En regroupant un modèle conceptuel et une de ses instances, on obtient la Base de Connaissances d'un projet déterminé.

6.3. PROBE: Prolog Object Extension

6.3.1. Présentation de PROBE

PROBE est un noyau orienté-objet construit au dessus de PROLOG. Il permet d'accéder à une Base de Connaissances, ainsi que de vérifier la cohérence de celle-ci. Ces opérations s'effectuent via un mecanisme de "propositions".

6.3.1.1. Outils fournis par PROBE

PROBE ne constitue pas à proprement parler un langage, mais est défini par six familles de predicats Prolog (permettant l'accès à la Base de Connaissances):

- création d'une Base de Connaissances.
- insertion d'éléments dans une Base de Connaissances déterminee.
- recherche d'éléments.
- suppression d'éléments.
- modification d'éléments.
- mise en oeuvre du mécanisme de "propositions" (présenté au point 6.3.1.3.).

On notera qu'un élément d'une Base de Connaissances manipulé par l'un de ces predicats est aussi bien une classe (définition de classe) qu'une instance particulière d'une classe.

6.3.1.2. Vérification de cohérence

Ce mécanisme est particulièrement important dans un système orienté-objet traitant l'héritage multiple et les contraintes définies par l'utilisateur (via les slots de contraintes).

PROBE essaye d'assurer que la Base de Connaissances évolue d'état cohérent en état cohérent (certains conflits étant parfois impossibles à traiter sans intervention de l'utilisateur).

PROBE vérifie la cohérence de la Base de Connaissances à deux niveaux:

- Le **niveau conceptuel** prend en compte la cohérence du modèle conceptuel de la Base de Connaissances, c'est-à-dire les définitions de classes, les relations entre classes.
- Le **niveau factuel**, quant à lui, prend en compte la cohérence d'une instance de ce modèle, c'est-à-dire les instances des classes.

Au niveau de cohérence conceptuel, trois types de cohérence sont vérifiés:

- La cohérence atomique:

  La vérification de la cohérence atomique est une simple vérification syntaxique pour la définition de chaque classe, sans aucune vérification des liens possibles entre les classes.

- La cohérence verticale:

  La vérification de la cohérence verticale ne porte que sur les liens IS-A définis entre les classes.

- La cohérence horizontale:

  La vérification de la cohérence horizontale porte sur les différentes relations (autres que les liens IS-A) entre les classes. Elle consiste notamment à vérifier qu'aucune définition de slot ne référence des classes ou des slots non existants.

Au niveau de cohérence factuel, on vérifie les types de cohérence suivants:

- La cohérence atomique et verticale:

  La vérification de la cohérence atomique et verticale est effectuée simultanément au niveau factuel. Elle vérifie, pour chaque instance particulière, la correction des valeurs des slots (ne référençant pas d'autres classes) par rapport à leur définition.

- La cohérence horizontale:

  La vérification de la cohérence horizontale porte sur les différentes relations (autres que les liens IS-A) entre les classes. Elle consiste donc à vérifer qu'aucune valeur d'un slot ne référence des instances inexistantes.

  Elle concerne également la vérification des contraintes définies dans le modèle conceptuel.

6.3.1.3. Le mécanisme de "propositions"

PROBE gère la vérification de cohérence de la Base de Connaissances au moyen d'un mecanisme de "propositions". Ce mecanisme peut être décrit comme suit.

Lors de l'utilisation d'un prédicat PROBE conduisant à une modification de la Base de Connaissances, cette modification n'est pas immediatement reportee sur son contenu actuel.

Au contraire, cette modification est gardée en memoire, et appelée "proposition". En général, cette proposition n'est générée que si aucune inconsistance n'est détectée au niveau de la cohérence atomique et verticale.

Ceci permet d'effectuer plusieurs modifications successives de la Base de Connaissances en vérifiant la cohérence atomique et verticale pour chaque changement pris séparément.

Ensuite, à la demande du programmeur, et si la cohérence est vérifiée, PROBE rend effectifs en mémoire centrale les propositions de modifications. Si une inconsistance est détectée, PROBE interroge le programmeur de manière à obtenir soit une décision de retour à l'état coherent actuel de la Base de Connaissances, soit une modification rétablissant la cohérence pour l'ensemble des propositions.

6.3.2. Concepts fondamentaux

Les concepts fondamentaux manipulés par PROBE sont:

- la Base de Connaissances:

  Elle comprend le modèle conceptuel, et une instance de ce modèle.

  La Base de Connaissances est vue comme une collection d'objets (définitions de classes et instances de classes). PROBE manipule sans distinction classes et instances.

- les classes:

  Toute classe est identifiée par un nom, et décrite par une énumération de slots et une appartenance à une et une seule métaclasse. On distingue les métaclasses suivantes:

  Primitive (classes integer, integer(m,n), real, string, string(m,n)),

  Enumerated (classes décrites par une liste d'atomes),

  Range (classes définies par un intervalle d'entiers ou réels),

  Entity (classes représentant des entités dont la valeur ne dépend pas de celles de ses attributs),

  Aggregate (classes comme produit cartésien de valeurs).

- les instances:

  Chaque instance d'une classe est une occurrence de la classe.

- les slots: caractérisés par une liste de facettes.

  Ces facettes sont les suivantes:

  - categ (correspondant à la facette category de TDL),
  - card (correspondant à la facette card de TDL),
  - def (définissant l'ensemble de valeurs possibles du slot, et décrite par une expression PCL bien formée),
  - reverse (notion plus générale que celle donnée par la facette inverse de TDL, dont la définition précise sera donnée dans la définition du modèle conceptuel de la Base de Connaissances pour le langage PCL),
  - default (correspondant à la facette default de TDL),
  - comment (correspondant à la facette comment de TDL).

- les relations IS-A:

  Le concept de spécialisation d'une relation IS-A est plus général que pour TDL: un slot S d'une sous-classe A (d'une classe B) peut être redéfini, sans pour autant être une restriction du slot S défini pour B.

6.4. PCL: Prolog Constraint Language:

6.4.1. Présentation de PCL

Le langage PCL doit permettre d'exprimer des contraintes et des requêtes sur la Base de Connaissances, ainsi que des définitions de slots:

- Une contrainte est une condition portant sur les éléments de la Base de Connaissances, et qui doit constamment être vérifiée.
- Une requête est une expression dont l'évaluation a pour effet de calculer l'ensemble des éléments (instances ou tuples d'instances) vérifiant une certaine condition.
- Une définition de slot permet de définir un slot du modèle conceptuel.

Ces expressions de base sont des cas particuliers d'expressions PCL, dont la forme générale est précisée au point suivant.

6.4.2. Forme générale d'une expression PCL

Le langage PCL est basé sur le calcul relationnel.

La forme générale d'une expression PCL est la suivante, avec n supérieur ou égal à 1:

SET ($[x_1, \ldots, x_n]$ **WHERE** $P(x_1, \ldots, x_n)$ **)**

Des formes secondaires peuvent en être dérivées, en omettant certains composants optionnels:

- **SET (** $[x_1, \ldots, x_n]$ **)**
- **(** $[x_1, \ldots, x_n]$ **WHERE** $P(x_1, \ldots, x_n)$ **)**
- **(** $[x_1, \ldots, x_n]$ **)**
- $P(x_1, \ldots, x_n)$

Dans ces expressions, $P(x_1, \ldots, x_n)$ est une formule construite selon le schéma ci-dessous.

Soient P_1, P_2 deux formules, alors:

- un atome est une formule
- NOT P_1 , (P_1) sont des formules
- P_1 AND P_2 , P_1 OR P_2 sont des formules
- Si $P_1(s)$ est une formule contenant la variable libre s:

EXIST s IN setExpr: $P_1(s)$

FORALL s IN setExpr: $P_1(s)$

sont des formules

Un atome est une formule élémentaire permettant d'exprimer:

- L'appartenance d'une valeur à un ensemble,
- L'inclusion d'un ensemble dans un autre,
- Une comparaison

Remarquons qu'un atome doit également pouvoir contenir des expressions arithmétiques.

7. Objectifs du mémoire

Les points précédents ont donc présenté le projet DAIDA, dans lequel s'inscrit ce mémoire, en détaillant les parties de ce projet auxquelles on fera référence dans la suite. Reste à préciser exactement l'objectif de ce mémoire.

Comme on l'a dit, son objectif fondamental est de construire le langage PCL. Cet objectif peut être décomposé comme suit:

- Définition précise et complète du méta-modèle de Base de Connaissances utilisé par PCL.

  En effet, cette définition, si elle existe actuellement de manière implicite dans un certain nombre de documents internes au projet ([MEIR-TR-VEN], [MEIR-TR]), n'est pas assez précise et organisée que pour servir de base à la construction d'un langage

- Définition du langage PCL.

  La seule définition existante du langage PCL est la description de la "Forme générale d'une expression PCL" qui a été présentée au point précédent. Cette description est assortie d'une première définition de la syntaxe du langage ([MEIR-TR-VEN], [MEIR-TR]).

  Ces éléments restent trop généraux et imprécis que pour être considérés comme une définition du langage.

- Implémentation du langage PCL.

  Cette implémentation sera réalisée en langage BIM-PROLOG.

Deuxième partie

Cadre informatique du mémoire

1. Introduction

La première partie a présenté le projet DAIDA, et établi les objectifs du mémoire. Dans cette première partie, on a pu constater que ce mémoire touchait à plusieurs domaines informatiques, parmi lesquels l'approche orientée-objet et la programmation logique.

Avant de se consacrer à la réalisation des objectifs du mémoire proprement dits, il semble utile de s'attarder quelque peu à ces deux domaines.

D'abord, on présentera l'approche orientée-objet, en essayant de souligner les caractéristiques qui font son originalité, et en précisant un certain nombre de concepts utiles pour la suite du travail.

Ensuite, on montrera en quoi cette approche peut être combinée avec la programmation logique, en vue d'obtenir des outils plus puissants.

La programmation logique, quant à elle, ne sera pas présentée plus en détail, car elle n'est utilisée dans ce memoire que comme outil d'implémentation. Pour plus de détails concernant cette approche, on peut par exemple consulter [CLOCK-MELL], [COE-COT], [DEV], [MARC] et [STER-SHAP], ainsi que [DR-N-R].

2. L'approche orientée-objet

2.1. Apparition de l'approche orientée-objet

L'apparition de cette approche apparaît comme une conséquence de la façon dont les informaticiens ont traditionnellement abordé la construction de systèmes informatiques.

Partant du fait que les petits problèmes sont plus aisés à résoudre que les gros, on a souvent eu recours à des méthodes de décomposition de type Top-Down.

Cependant, l'application aveugle de ces méthodes conduit à la définition d'un nombre considérable de structures de données et de programmes très spécialisés, propres à la résolution de petits sous-problèmes restreints. Cette situation a les désavantages suivants:

- Lourdeur d'implémentation d'un système, vu le grand nombre de composants qu'il contient.
- Caractère peu évolutif d'un système, vu le grand nombre de petites modifications à lui apporter lors de changements qui peuvent sembler globalement mineurs.
- Absence de réutilisabilité des composants des systèmes, obligeant l'informaticien a resoudre, pour chaque nouveau système, un grand nombre de problèmes qu'il a déjà "presque" résolus dans ses produits précédents.

En réaction contre ces inconvénients, les défenseurs de l'approche orientée-objet proposent d'adopter une méthode d'analyse de type Bottom-Up.

Cette démarche consiste dans un premier temps à définir un certain nombre d'objets ou types abstraits, fournissant des structures de donnees relativement générales, assorties des fonctions utiles à leur manipulation. Ensuite, ces objets sont assemblés de façon à construire un système particulier, aisément modifiable.

On notera cependant que cette approche, poussée à l'extrême, presente les inconvénients suivants:

- Il semble difficile d'établir a priori la liste des données et des fonctions qui seront utiles à la construction d'un système (et a fortiori celles qui seront utiles lors de son évolution, ou lors de la construction des systèmes suivants).
- De plus, la construction d'un objet déterminé, sans référence à l'usage précis que l'on veut en faire, risque de conduire à des types abstraits tellement généraux qu'ils compliqueraient la construction d'un système, plutôt que de la simplifier.

En fin de compte, il semble plus raisonnable d'adopter une approche nuancée, combinant les avantages des deux précédentes.

Elle consisterait, dans un premier temps, à recourir à une analyse Top-Down, moins détaillée que de coutume, pour déterminer les fonctionalités générales du système à construire. Ensuite, une analyse Bottom-Up, guidée par les lignes directrices ainsi établies, permettrait de déterminer les objets réellement necessaires au système. Ces objets seraient ensuite définis de manière suffisamment générale que pour assurer leur réutilisabilité. Leur assemblage final fournirait alors le système désiré.

2.2. Caractéristiques générales de l'approche orientée-objet

L'approche orientée-objet utilise un certain nombre de principes d'abstraction, qui sont d'ailleurs utilisés couramment dans d'autres domaines. Le point suivant présente ces principes de manière générale.

2.2.1. Principes d'abstraction utilisés

Ces principes d'abstraction sont les suivants:

- La **classification/instanciation**:

  Cette abstraction permet de distinguer une description de type (définition d'objet) et une occurrence de ce type (objet ou instance de la définition).

- L'**agrégation/décomposition**:

  Cette abstraction permet de considérer un objet sans mentionner les différents composants conceptuels de cet objet.

- La **généralisation/spécialisation**:

  Cette abstraction permet d'organiser les différents objets décrits en hiérarchies, en introduisant le concept d'héritage.

La distinction entre ces trois principes fournit un moyen commode de présenter les caractéristiques générales de l'approche orientée-objet. Les points suivants vont détailler la façon dont ces principes sont généralement mis en oeuvre dans l'approche orientée-objet. Pour chacun d'eux, on précisera:

- les concepts qui leur correspondent,
- les autres concepts, dont la définition repose sur celle des précédents.

2.2.2. La classification/instanciation:

Ce mécanisme d'abstraction permet donc de distinguer la définition de l'objet de l'objet lui-même.

La définition de l'objet fait partie du modèle conceptuel, et est généralement appelée **classe**, **frame**. Il s'agit généralement d'une description des caractéristiques d'un objet (description de type).

L'objet manipulé à l'exécution par le système est une représentation d'un objet du monde réel. Il satisfait la définition qui en a été faite dans le modèle conceptuel. Un objet est une **instance** d'une classe, frame. On parlera également d'**entité**.

La classification consiste donc a grouper les entités en une seule classe. Au contraire, l'instanciation permet d'obtenir une entité conforme à la définition de l'objet (à la classe).

2.2.3. L'agrégation/décomposition:

Ces deux principes d'abstraction lient la définition d'un objet et ses caractéristiques.

Ces principes sont appliqués lorsqu'il s'agit de définir une classe d'objets, en décrivant les caractéristiques ou propriétés communes a toutes ses instances. Une caractéristique d'un objet est appelée, selon les cas, un **attribut**, un **slot**.

Ces principes sont également appliqués à un autre niveau. Certains langages de programmation orientés-objet permettent une classification des différentes classes définies dans le modèle conceptuel selon les fonctionnalités communes à ces classes et selon le type de traitement qui peut leur être appliqué. Cette classification définit le concept de **métaclasse**, qui peut être vu comme un concept générique de classe (une métaclasse est définie comme une classe de classes)[1].

A partir du concept d'attribut défini ci-dessus, on peut mettre en évidence d'autres particularités de l'approche orientée-objet.

Les attributs peuvent être de nature très différentes, et toutes n'existent pas dans chaque approche orientée-objet. Un attribut peut définir:

1. une valeur simple (réel, chaîne de caractères,...),

---

[1] Des langages comme EIFFEL ([MEY]) ne possèdent pas ce concept de métaclasse. Cette notion est également absente des approches basées sur les frames, ou les réseaux sémantiques

2. une référence ("pointeur" vers un objet associé, contraintes sur la nature de cet objet),

3. une fonction ou une procédure,

4. une propriété de classe, partagée par toutes les instances de celle-ci[1] ,

5. une propriété définissant un invariant, une pré- ou post-condition (contrainte qui sera vérifiée en à tout moment de l'existence de l'instance, à sa création, à sa destruction)[2].

Ces différentes caractéristiques permettent également d'introduire la notion de **communication/coopération** entre les objets. Cette notion est toujours présente, mais sous des formes qui ne sont pas aussi explicites dans tous les langages.

Des langages tels que Smalltalk ([SMALL]) et EIFFEL ([MEY]) offrent un mécanisme où toute coopération d'objets à la réalisation d'une tâche (execution d'une procédure) se fait par le biais d'envoi de messages entre les objets

Le langage PCL est, quant à lui, un langage de type statique (consultation d'une Base de Connaissances), qui ne prévoit pas de mécanisme d'appel de procédures. Cependant, il introduit une certaine forme de coopération entre les objets par la notion de classe client (introduite via les attributs de type référence). C'est également le cas des langages basés sur les réseaux sémantiques.

2.2.4. La généralisation/spécialisation:

La spécialisation permet d'introduire un certain nombre de détails additionnels dans une description d'un sous-ensemble, alors que la généralisation a pour but de rassembler dans une description unique les caractéristiques communes à différents sous-ensembles.

Cette notion donne lieu au principe de répartition des classes en une structure hiérarchique de classes et sous-classes. Cette relation de classe à sous-classe est généralement appelée **relation IS-A**.

---

[1] Par exemple, un attribut "le_plus_grand_rectangle" est une propriété de l'ensemble des instances de la classe "rectangle" (langage KEE [FIK-KEH])

[2] On retrouve de tels attributs dans les langages EIFFEL, RML ([GREEN]). PCL offre aussi une telle facilité, mais sous la forme d'une contrainte (expression booléenne).

Comme conséquence de la hiérarchisation des classes, la notion d'**héritage** est apparue (dans un souci de gain de place et de réduction de la répétition d'une même information).

Cette notion d'héritage est très générale, et est appliquée différemment dans les nombreux langages. Il est donc plus aisé de présenter les différentes manières de traiter l'héritage, plutôt que de tenter de fournir une classification des langages selon le type d'héritage.

L'héritage est généralement associé à la notion d'extension. Une classe hérite des propriétés générales de sa super-classe, et possède des propriétés qui lui sont propres.

Cet héritage peut être:

- strict (les propriétés sont héritées comme telles, sans modification),
- associé à la notion de spécialisation (la définition d'un attribut peut être une restriction de la définition du même attribut dans la super-classe),
- associé à la notion de redéfinition (la définition d'un attribut peut être une redéfinition du même attribut dans la super-classe).

Il faut encore envisager le cas de l'héritage multiple. En effet, une classe peut hériter de plusieurs classes parents.

L'héritage peut occasionner des conflits dans le cas où un même attribut est hérité de plusieurs super-classes. Ce type de conflit peut être résolu de manières diverses:

- l'attribut n'est pris qu'une seule fois en compte, s'il possède exactement la même définition dans les deux super-classes,
- l'attribut est pris plusieurs fois en compte, mais avec des noms différents permettant d'indiquer de quelle classe il est hérité,
- le système exige que l'attribut soit redéfini complètement au niveau de la classe qui en hérite.

3. Approche orientée-objet et programmation logique

Au vu de l'intérêt actuellement porté à l'approche orientée-objet ainsi qu'à la programmation logique, on peut se demander s'il n'est pas possible de combiner ces deux techniques, dans l'espoir de creer des outils très performants rassemblant les avantages de chacune d'elles.

De tels travaux ont été réalisés, et l'on peut dégager quelques voies de recherche qui ont été suivies.

3.1. L'extension du langage de programmation logique

3.1.1. La notion d'héritage

L'un des concepts primordiaux de l'approche orientée-objet est la notion d'héritage. Certains ont proposé d'ajouter cette notion aux langages de programmation logique.

C'est le cas du langage LOGIN [A.K.-NASR]. LOGIN est un langage de type PROLOG, où la notion de type est introduite, permettant de définir explicitement des hiérarchies de types, ce qui accélère le processus de résolution. Le langage résultant est une généralisation de PROLOG. Un programme PROLOG ne doit donc pas être modifié pour appartenir au langage LOGIN, et il s'agit là d'un avantage non négligeable pour le programmeur.

Le langage EPOS ([HUB-VAR]) constitue lui aussi une extension de PROLOG, où la hiérarchisation des types peut s'exprimer par des relations d'ordre sur les types.

3.1.2. La communication entre objets

Cette caractéristique de l'approche orientée-objet est disponible dans un langage tel que Concurrent Prolog, développé par E. Shapiro ([SHAP]). Elle est une conséquence du fait que ce langage intègre la notion de parallélisme, qui requiert la possiblité de faire communiquer certains objets afin de synchroniser leur comportement.

Ce langage permet donc de représenter facilement des systèmes d'objets cooperants, mais ne prend pas en considération la notion d'héritage.

3.2. L'ajout d'un "package" orienté-objet au langage de programmation logique

Dans cette optique, le langage de programmation logique n'a pas été modifié. Une librairie de primitives supportant les notions de l'approche orientée-objet a seulement été ajoutée.

La faisabilité de cette approche a déjà été démontrée pour d'autres langages, tels LISP (langages Loops, Flavors) et C (C++, Objective-C) [BOBR], [MOON], [STROU], [COX].

On peut citer à cet effet les travaux de C. Zaniolo ([ZAN]). Son approche fournit toutes les caractéristiques importantes de la programmation orientée-objet, et est facilement implémentable pour tous les PROLOG existants.

3.3. L'utilisation de concepts orientés-objet dans le langage de programmation logique

Contrairement aux autres orientations déjà présentées, cette approche ne conduit à aucun langage nouveau. Elle consiste pour le programmeur à appliquer la méthodologie orientée-objet pour la conception de son système, et ensuite à programmer dans le langage (de programmation logique) à sa disposition les concepts lui permettant d'implémenter le système qu'il a décrit.

3.4. Utilisation de PROLOG comme outil d'implémentation

On citera finalement la technique qui consiste à utiliser un langage de programmation logique comme outil d'implémentation d'un langage orienté-objet. Cependant, cette technique ne constitue pas à proprement parler une intégration des concepts des deux méthodes.

Le langage de programmation logique (PROLOG) est utilisé ici comme outil d'implémentation d'un environnement orienté-objet, comme on aurait pu le faire avec tout autre langage de programmation.

On remarquera cependant que la puissance et la souplesse de PROLOG rendent aisée une telle implémentation.

C'est dans cette optique que s'inscrit l'environnement du prototypage du projet DAIDA, et plus particulièrement les langages PROBE et PCL.

Troisième partie

Théories et méthodes

Les deux premières parties ont défini les objectifs du mémoire, et ont précisé le cadre informatique dans lequel il se situe.

Cette troisieme partie constitue la base theorique sur laquelle reposent la définition et l'implémentation du langage PCL.

Elle présente de manière générale les différents problèmes auxquels on doit faire face lors de la construction d'un langage de ce type. Dans la mesure du possible, elle propose les différentes solutions applicables dans chaque cas, et détaille leurs avantages et leurs inconvénients.

Les trois grands domaines auxquels on s'intéressera sont la définition d'un méta-modèle de Bases de Connaissances, la définition d'un langage formalisé, et l'implémentation d'un tel langage.

Chapitre 1

Définition d'un méta-modèle de Bases de Connaissances

1. Le contexte de définition d'un langage

Une étape préliminaire à la définition de tout langage informatique est la description d'un contexte, d'un cadre précis dans lequel la définition du langage s'intègrera.

Dans le cas de langages de programmation "classiques", la définition explicite d'un tel contexte peut paraître superflue aux "programmeurs chevronnés". Notons cependant que cette définition peut malgré tout faciliter la compréhension d'un certain nombre de constructions du langage.

Dans le cas de langages moins classiques, tels les langages orientés-objets ou de programmation logique, les auteurs n'hésitent pas à présenter une description de leur approche avant de définir le langage lui-même ([MEY] - [STER-SHAP]).

Dans le cas des langages de requêtes tels que PCL, une telle définition prend tout son sens, comme on va le voir au point suivant.

2. Cas des langages de requêtes

Un tel langage permet de formuler des requêtes sur des Bases de Connaissances (ou "de Données"). Dans ce type d'environnement, il est important de bien distinguer deux grandes familles de concepts, afin d'éviter toute confusion:

1. Les concepts mis en oeuvre dans toutes les Bases de Connaissances qui peuvent être construites dans l'environnement du langage.

   Par exemple, une Base de Connaissances est toujours constituée d'un ensemble de classes, chacune d'elles étant définie par un certain nombre de slots.

2. Les concepts propres à une Base de Connaissances particulière, c'est-à-dire ceux qui sont définis par son modèle conceptuel.

   Par exemple, une Base de Connaissances utilisée dans le domaine de la production de logiciels peut définir des concepts tels que "programme", "spécification", ...

Le contexte de définition d'un langage de requêtes sera constitué de concepts du type 1 ci-dessus. Ce contexte devra être défini avec soin, afin d'éviter toute confusion entre les deux familles de concepts.

Un tel contexte définit en fait un "méta-modèle" des Bases de Connaissances pouvant être manipulées par le langage de requêtes. Ce "méta-modèle" devrait posséder les caractéristiques suivantes:

- Présentation exhaustive des concepts manipulés.
- Structuration de ces concepts facilitant leur compréhension ainsi que la définition du langage.
- Prise en compte, pour chaque concept, des aspects syntaxiques et sémantiques qui lui correspondent.

Le point suivant propose une méthode systématique permettant de définir un "méta-modèle" possédant ces caractéristiques.

3. Méthode de définition

3.1. Une méthode possible

La première démarche qui vient à l'esprit consiste à énumérer tous les concepts du modèle, en donnant pour chacun d'eux une définition syntaxique et sémantique. Cette méthode possède les inconvénients suivants:

- Une simple énumération de concepts, sans structure plus détaillée, n'est pas très pratique à l'usage. En effet, la recherche d'un concept particulier peut se révéler fastidieuse.
- De nombreux concepts ne se suffisent pas à eux-mêmes, mais sont définis en termes d'autres concepts du modèle. Dans ce cas, deux problèmes surgissent:
  - L'énumération doit éviter autant que possible les "références en avant".
  - Si certaines définitions sont "circulaires", il est impossible de les présenter sous forme d'une énumération (notamment, sans "références en avant").

Au vu de ces inconvénients, il nous a semblé utile d'élaborer une autre méthode, présentée au point suivant.

3.2. La méthode proposée

Cette méthode a été mise au point dans le but de remédier aux désavantages exposés au point précédent, et principalement à celui des définitions circulaires.

Elle propose de décomposer la définition d'un méta-modèle conceptuel en une hiérarchie de niveaux, appelés "niveaux de cohérence". A chacun de ces niveaux, on définira un certain nombre de concepts. Cette définition couvrira les aspects syntaxiques et sémantiques associés à chacun d'eux.

Cette structure hiérarchique est caractérisée par les propriétés suivantes:

- Tout niveau regroupe un ensemble de définitions de concepts.
- Le niveau le plus bas rassemble les définitions élémentaires des éléments de base pouvant être utilisés dans les modèles conceptuels correspondant au méta-modèle décrit.

  Il est à noter que ces définitions seront généralement raffinées dans les niveaux supérieurs.
- Chaque niveau doit, au minimum, contenir les concepts du niveau qui lui est strictement inférieur.

- L'existence de chaque niveau est justifiée soit par la **définition** de concepts nouveaux, soit par la **redéfinition** de concepts d'un niveau inférieur.

- Toute **définition** d'un concept ne peut référencer que des concepts déjà définis.

- Toute **redéfinition** d'un concept déjà défini dans le niveau inférieur doit être un raffinement (restriction) de la définition par l'ajout de conditions supplémentaires.

Etant donné un méta-modèle défini en suivant cette méthode, on dira qu'un modèle conceptuel donné est:

- cohérent syntaxiquement avec le niveau X du méta-modèle si tous ses concepts satisfont les définitions syntaxiques de niveau X du méta-modèle,

- cohérent sémantiquement avec le niveau X du méta-modèle si tous ses concepts satisfont les définitions sémantiques de niveau X du méta-modèle,

Enfin, on tiendra compte des remarques suivantes:

- Un modèle syntaxiquement (resp. sémantiquement) cohérent avec un certain niveau peut ne plus être cohérent avec un niveau supérieur, si ce niveau contient des concepts qui ont été redéfinis par des conditions supplémentaires plus fortes.

- En revanche, un modèle syntaxiquement (resp. sémantiquement) cohérent avec un certain niveau sera toujours cohérent avec tous les niveaux inférieurs.

Chapitre 2

Définition d'un langage formalisé

1. Introduction

Ce second chapitre traite le problème de la définition d'un langage formalisé. Il propose pour cela une démarche générale, et en détaille les différentes étapes.

2. Présentation de la démarche générale

La démarche que nous présentons dans les points suivants est une synthèse réalisée à partir des travaux de [KNU], [LECH], [LIV] et [TENN].

2.1. Identification des concepts du langage

La définition d'un langage formalisé pose tout d'abord le problème du choix des concepts que l'on veut prendre en compte.

Afin d'effectuer ce choix, il est nécessaire de disposer d'une notion intuitive du langage désiré. Ensuite, ayant déterminé intuitivement quels seront les concepts de base du langage, il conviendra d'en donner une définition précise.

Par exemple, dans le cas d'un langage de programmation, on pourra désirer disposer des notions suivantes:

- Types d'instructions possibles: la séquence, la boucle, la condition, l'appel de procédure, ...
- Types de données manipulées: les entiers et les chaînes de caractères.
- Types d'objets disponibles: les variables et les constantes.
- ...

2.2. Syntaxe et sémantique

Lorsque l'on a déterminé un concept, il s'agit de définir tant sa forme que sa signification. On distinguera à cet effet la syntaxe et la sémantique d'une construction.

La syntaxe précise la forme, la structure "compositionnelle" des constructions d'un langage. La sémantique, quant à elle, précise la signification qui leur est attribuée.

De cette façon, un langage formalisé peut être défini en deux étapes:

- définition de l'ensemble des expressions syntaxiquement correctes du langage (les "expressions bien formées"),
- définition de la manière dont on attribue un sens à une expression bien formée, ce qui divise cet ensemble en deux:
  - les expressions syntaxiquement correctes ayant une signification,
  - les expressions syntaxiquement correctes n'ayant aucun sens.

Par exemple, dans le cas d'un langage de programmation, l'instruction d'itération peut être définie comme suit:

- Du point de vue syntaxique:

  while condition_booléenne do instruction

- Du point de vue sémantique:

  "Evaluer condition_booléenne, puis, si elle a la valeur VRAI, exécuter instruction et recommencer, sinon, passer à l'instruction suivante"

Notons à ce propos que la distinction entre la syntaxe et la sémantique n'est pas propre à la définition d'un langage informatique, mais est d'application dans tout domaine manipulant des systèmes formels ou des représentations d'objets ([HOFST], [MEY]).

2.3. Syntaxe abstraite et syntaxe concrète

Classiquement, la syntaxe d'un langage est présentée comme étant constituée d'un alphabet, ainsi que d'un ensemble de règles permettant de définir l'ensemble des expressions bien formées du langage.

Ce type de syntaxe contient généralement un certain nombre de détails, liés aux représentations choisies, qui non seulement ne présentent aucun intérêt du point de vue de la perception des mécanismes de formation d'expressions, mais en plus, compliquent leur compréhension.

La notion de syntaxe abstraite a été proposée pour remédier à ces inconvénients. Son but est de présenter les mécanismes essentiels du langage, en limitant dans la mesure du possible l'introduction de détails inutiles[1]. De cette façon, le langage peut être

---

[1] Evidemment, ce but ne pourra jamais être totalement atteint: dès que cette syntaxe sera exprimée dans un langage quelconque, certains détails de représentation devront être introduits. Le principal est d'essayer de rester "le plus abstrait possible".

complètement défini sans aucune référence à un choix de représentation.

Par rapport à cette syntaxe abstraite, on définit alors une syntaxe concrète du langage, qui introduit les détails additionnels propres à la représentation effective de ce langage.

L'intérêt d'une telle approche est évident. Elle facilite la définition d'un langage en permettant de distinguer clairement:

- la définition de sa sémantique, qui peut être réalisée complètement sur base de la syntaxe abstraite,
- la prise en compte de tous les détails syntaxiques, qui sont introduits (dans des proportions variables selon les cas) afin:
  - de faciliter la lecture du langage par un être humain,
  - de faciliter la construction d'un analyseur syntaxique.

Par exemple, dans le cas d'un langage informatique, l'instruction conditionnelle peut être définie:

- du point de vue abstrait, comme une relation ternaire:

  condition(expression_booléenne,instruction_1,instruction_2),

  permettant d'exprimer clairement le rôle des trois éléments de la relation, sans référence à leur représentation exacte;
- du point de vue concret, comme une construction possédant une des formes suivantes:

  if condition_booléenne then instruction_1 else instruction_2 endif

  if condition_booléenne then instruction_1 endif

  où le mot endif a été introduit afin d'éviter les problèmes d'ambiguïté.

En outre, on remarquera que rien n'empêche, étant donné une syntaxe abstraite, de définir plusieurs syntaxes concrètes, correspondant à plusieurs représentations du langage.

Par exemple, un langage de programmation peut posséder une représentation "visuelle" sous forme d'organigrammes que l'on édite avec un interface graphique approprié, et une représentation plus classique, que l'on édite avec un traitement de texte. C'est le cas notamment des langages VIP et 4D, qui sont utilisés sur Apple® Macintosh™.

2.4. Récapitulation

Les considérations précédentes permettent de dégager quatre étapes fondamentales dans la définition d'un langage:

- identification des concepts
- définition d'une syntaxe abstraite
- définition de la sémantique
- choix d'une ou plusieurs syntaxes concrètes, qu'il ne reste plus qu'à implémenter.

Les points suivants détaillent chacune de ces étapes.

3. Identification des concepts du langage

Cette étape consiste à déterminer, de manière intuitive, quels seront les concepts du langage. Sur base de ces concepts, on définira les constructions élémentaires et les constructions composées du langage.

Les constructions élémentaires seront simplement énoncées, tandis que les constructions composées seront exprimées en fonction des constructions élémentaires en précisant les mécanismes de composition utilisés.

4. Syntaxe abstraite et grammaires attribuées

Partant d'une description intuitive des concepts de base du langage, cette étape consiste à définir aussi abstraitement que possible, une syntaxe ayant pour but de faciliter la définition de la sémantique de toute expression. Elle doit permettre de déterminer sans ambiguïté l'ensemble des expressions bien formées du langage.

4.1. Definition de la syntaxe abstraite

Cette définition peut être réalisée de diverses manières, selon la complexité du langage considéré.

On peut par exemple utiliser la langue naturelle, augmentée de notations appropriées. Cependant, la lourdeur d'utilisation de cette méthode la rend vite inutilisable pour des problèmes complexes. On peut également utiliser un formalisme de type fonctionnel, où les constructions sont représentées par des fonctions.

Le plus souvent cependant, on préfère utiliser un formalisme de type BNF (Backus Naur Form).

Bien que ce type de formalisme soit généralement utilisé pour définir la syntaxe concrète d'un langage, il est possible de lui donner une interprétation différente, permettant de définir une syntaxe abstraite. La grammaire abstraite d'un langage est alors constituée d'un ensemble de règles de la forme:

$$X ::= Y_1, Y_2, Y_3 \ \ldots\ldots\ , Y_n$$

Cette règle définit la construction syntaxique X comme un n-uplet constitué des constructions $Y_1, Y_2, Y_3 \ \ldots\ldots\ , Y_n$.

Ces constructions sont soit primitives, soit composées. Dans le premier cas, elles appartiennent à des "domaines syntaxiques primitifs" du langage (entier, réel, chaîne de caractères, ...). Dans le second cas, elles sont elles-mêmes définies par d'autres règles de la grammaire.

Lorsqu'une construction peut être définie par plusieurs n-uplets différents, on peut soit écrire plusieurs règles:

$$X ::= Y_1, Y_2$$

$$X ::= Y_1, Y_2, Y_3$$

soit les regrouper en une seule:

$$X ::= Y_1, Y_2 \mid Y_1, Y_2, Y_3$$

Il est à noter que dans le cas où le langage à définir possède des caractéristiques complexes qui ne peuvent être exprimées facilement dans ce formalisme, rien n'empêche de l'enrichir en introduisant des outils appropriés[1].

Une fois la syntaxe abstraite définie, il reste à décrire le mécanisme permettant de déterminer si une construction donnée constitue une expression abstraite syntaxiquement correcte. Le point suivant présente une façon de définir ce mécanisme.

4.2. Correction syntaxique d'une expression abstraite

4.2.1. Méthode choisie

Dans le cadre de ce mémoire, cette correction a été vérifiée par la méthode des grammaires attribuées.

Il s'agit d'un procédé uniforme, indépendant du langage formel considéré, dont l'utilité ne se limite d'ailleurs pas à la vérification de la correction syntaxique d'une expression abstraite. On peut en effet l'utiliser également:

- pour convertir une grammaire abstraite en une grammaire concrète,
- pour définir la sémantique d'une expression (abstraite ou concrète) syntaxiquement correcte,

et, en général, dans tous les cas où il s'agit de définir formellement des propriétés d'un langage défini par une grammaire.

4.2.2. Présentation de la méthode

Cette méthode peut donc être utilisée lorsque l'on désire obtenir des informations à propos des constructions syntaxiques correspondant à une grammaire donnée.

Pour obtenir ces informations, on définit un certain nombre de fonctions, appelées attributs, dont on définit la valeur pour les différentes règles de la grammaire. Ces fonctions sont de la forme:

---

[1] Par exemple, les tableaux de compatibilité de types utilisés pour PCL, décrits en annexe 5.

attr: **CONS** ----------> **RES**

expr ~~~~~~~~> attr(expr)

où **CONS** est un ensemble de constructions syntaxiques de la grammaire, et **RES** est l'ensemble des valeurs possibles de l'attribut, c'est-à-dire l'ensemble des valeurs possibles d'une des informations à laquelle on s'intéresse.

Par exemple:

- dans le cas où l'information recherchée est de savoir si une construction syntaxique est correcte ou non, l'ensemble **RES** peut être défini comme un ensemble à deux valeurs "correct" et "non-correct".

  L'attribut attr permet alors d'attribuer à toute construction la valeur "correct" si l'expression est syntaxiquement correcte, et la valeur "non-correct" dans le cas contraire.

- Dans le cas où l'information recherchée est l'expression concrète correspondant à une expression abstraite, l'ensemble **RES** est défini comme l'ensemble des expressions concrètes.

  L'attribut attr permet alors d'obtenir l'expression concrète correspondant à une expression abstraite donnée.

4.2.3. Attributs hérités et synthétisés

On distingue deux types d'attributs: les attributs hérités et les attributs synthétisés. Afin de faciliter leur présentation, fixons les notations suivantes:

- Soit G, la grammaire définissant un langage donné.
- Soit R, une règle grammaticale de G, de la forme:

  $Y ::= X_1, X_2, ..., X_n$,

  qui définit la construction Y comme étant composée des constructions $X_1, X_2, ..., X_n$.
- Soit A, un attribut défini sur la grammaire G. Soit A(C), la valeur de cet attribut pour une construction C quelconque.

L'attribut A peut être considéré de deux façons différentes au niveau de la règle R.

- Soit, on connaît la valeur A(Y), et, pour un composant X_i de R, la valeur $A(X_i)$ est définie comme une fonction de A(Y)[1].

  On dit alors que A est un attribut hérité par la règle.

- Soit, la valeur A(Y) est définie comme une fonction des valeurs A(X1), A(X2), ..., A(Xn)[2].

  On dit alors que A est un attribut synthétisé par la règle.

A titre d'exemple, considérons un langage de programmation que l'on désire compiler, et dans lequel les variables d'un programme doivent être déclarées en tête de celui-ci. On pourra disposer des définitions suivantes:

- Une grammaire, contenant notamment la règle:

  Programme ::= Declarations, Corps_de_programme

- Un attribut Decl, dont la valeur représente l'ensemble des variables déclarées dans le programme

- Un attribut Trad, dont la valeur représente le programme compilé.

Au niveau des différentes règles de la grammaire, ces attributs seront utilisés de la façon suivante:

- L'attribut Decl :
  - sera synthétisé par l'ensemble des règles correspondant aux Declarations,
  - sera hérité par l'ensemble des règles correspondant à un Corps_de_programme.
- L'attribut Trad, quant à lui, sera synthétisé par l'ensemble des règles correspondant au Corps_de_programme, vraisemblablement en utilisant la valeur de l'attribut Decl.

Ce mécanisme d'héritage et de synthèse des attributs peut être illustré par le schéma suivant:

---

1 Cette valeur peut également dépendre d'autres paramètres, comme par exemple la valeur d'autres attributs.

2 De la même façon, cette valeur peut dépendre de paramètres supplémentaires.

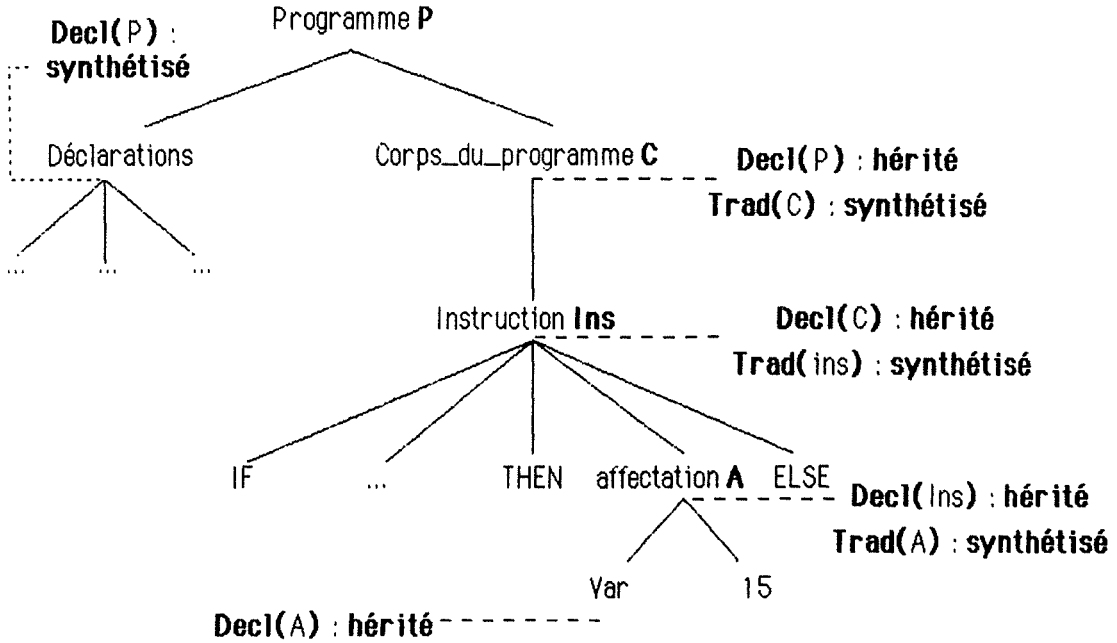

Figure 4 : Mécanisme d'héritage et de synthèse des attributs

4.2.4. Exemple concret

A titre d'exemple, voici une grammaire attribuée permettant de définir la valeur numérique d'une suite binaire.

La grammaire abstraite est définie par les règles de production suivantes:

N ::= B
B ::= C | B C
C ::= 0 | 1

Le problème du calcul de la valeur d'une suite binaire peut se généraliser à celui du calcul de la valeur d'une "sous-suite" binaire, faisant partie d'une suite binaire plus grande N .

On définit un attribut synthétisé v:

v: **B + C** ----------> **RES**

ssbin ~~~~~~~~> v(ssbin)

où v(ssbin) est la valeur numérique de ssbin en tant que préfixe d'une suite binaire N .

On définit également un attribut hérité s:

s: **B + C** ----------> **RES**

ssbin ~~~~~~~~> s(ssbin)

où s(ssbin) est la longueur du suffixe ajouté à ssbin pour former la suite binaire N (le nombre de zéros ajouté à ssbin).

Ces attributs peuvent être illustrés par la figure suivante:

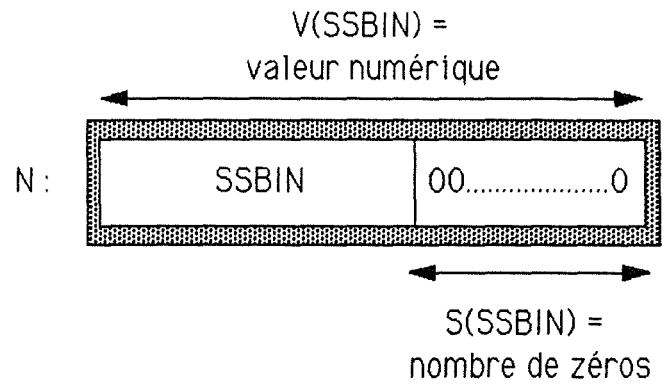

Figure 5 : Attributs définissant la valeur d'une suite binaire

Il reste à définir la valeur des deux attributs pour chaque règle de production.

Pour N ::= B:	v(N) = v(B) s(B) = 0
Pour B ::= C :	v(B) = v(C) s(C) = s(B)
Pour B ::= B' C:	v(B) = v(B') + v(C) s(B') = s(B) + 1 s(C) = s(B)
Pour C ::= 0:	v(C) = 0 s(0) = s(C)
Pour C ::= 1:	$v(C) = 2^{s(C)}$ s(1) = s(C)

5. Sémantique

Lorsque l'on parcourt la littérature concernant les langages de programmation, on constate qu'il existe une certaine difficulté, pour l'informaticien, d'exprimer clairement la sémantique des langages qu'il définit.

Avant de présenter quelques méthodes de définition de la sémantique, il est donc utile de s'interroger sur l'intérêt cette définition.

5.1. Intérêt de la définition de la sémantique

En général, la définition d'un langage de programmation comporte trois aspects ([LIV]):

- La définition formelle de la majeure partie de la syntaxe (concrète) du langage.

  Cette définition formelle est généralement restreinte à ce qui peut s'exprimer sous forme de grammaires "context-free".

- La définition moins formalisée de conditions syntaxiques supplémentaires ne pouvant s'exprimer dans une grammaire de type "context-free".

- Lorsqu'elle existe, une définition peu formalisée de la sémantique.

Cette dernière définition est donc souvent négligée. Pourtant, un certain nombre de raisons justifient son intérêt:

- Du point de vue de l'implémentation du langage, une bonne définition de la sémantique évite que deux implémentations différentes n'occasionnent deux interprétations différentes d'un même programme.

- Du point de vue de la programmation dans ce langage, une bonne définition de sa sémantique permet une meilleure maîtrise de celui-ci.

- Du point de vue de la conception de langages nouveaux, une bonne définition de la sémantique doit permettre une normalisation plus facile, ainsi qu'une comparaison aisée de différents langages.

Dans cette optique, et vu la diversité des buts poursuivis, une seule méthode de définition de la sémantique risque de ne pouvoir rencontrer tous les objectifs. Il est plus intéressant de pouvoir donner plusieurs définitions compatibles d'un même langage, chacune s'adressant à une catégorie particulière d'utilisateurs. Le point suivant détaille quelques-unes des méthodes possibles.

5.2. Méthodes de définition de la sémantique

5.2.1. Définition par la syntaxe

L'argument avancé par les utilisateurs de cette technique est qu'une syntaxe (concrète) suffisamment claire et bien choisie se suffit à elle-même pour définir la signification d'un langage.

Cette façon de faire revient à ne pas définir de sémantique du tout, ce qui, on l'a vu au point précédent, présente plusieurs désavantages.

De plus, les choix de représentation introduits dans la définition syntaxique concrète peuvent conduire à des interprétations différentes selon les utilisateurs.

5.2.2. Définition par l'implémentation

D'autres diront qu'une fois le langage implémenté, le programme ainsi écrit peut servir de définition à la sémantique.

A nouveau, cette solution ne peut être satisfaisante, notamment pour les raisons suivantes:

- Etant donné la façon dont certains programmeurs écrivent leurs programmes, il n'est pas toujours évident que le texte d'un programme permette de comprendre ce qu'il fait réellement.
- De plus, même un programme d'une clarté exemplaire ne fait que manipuler des représentations arbitraires d'objets, souvent au moyen de nombreux artifices de programmation, qui ne peuvent que difficilement éclairer quant à son utilité réelle.
- Enfin, cette technique va à l'encontre des méthodologies prônant la modularité et la réutilisabilité, qui sont basées sur l'indépendance des choix de représentation et d'implémentation, l'abstraction de données, la minimisation de connaissances superflues ...

5.2.3. Définition en langage naturel

Cette approche a au moins le mérite de reconnaître l'utilité d'une définition explicite de la sémantique.

Cependant, on lui attribue habituellement de nombreux désavantages, tels que les possibilités de redondance, de contradiction, de références en avant, d'imprécisions, d'ambiguïtés, d'oublis, ...

Il faut noter que ces désavantages ne sont pas propres au langage naturel. Il est en effet possible de produire des textes formels ayant les mêmes défauts. En dernier ressort, seuls le soin et la rigueur permettent de donner une définition satisfaisante de la sémantique, quel que soit le type de langage utilisé.

Bien plus, il est parfois préférable de se limiter à une description précise et complète en langage naturel, plutôt que d'introduire à outrance des notations formelles dans l'espoir de "donner plus de poids" à une description imparfaite.

5.2.4. Définition en langage formel

Dans [LIV], on peut trouver une classification assez schématique des différentes méthodes formelles permettant de définir la sémantique d'un langage de programmation.

Le premier type d'approche est la définition interprétative ou opérationnelle de la sémantique. Elle consiste à définir une *fonction sémantique* associant à un couple (programme, donnée) le résultat d'un calcul (résultat de l'exécution du programme avec les données fournies).

Le second type d'approche est l'approche dénotationnelle. Elle possède les caractéristiques essentielles suivantes:

- A chaque construction du langage est associée une fonction.
- La signification d'une construction composée est définie en fonction de la signification de ses composants.

Dans le cas d'un langage de programmation, on associe à tout programme une fonction définie de l'ensemble des données de ce programme, vers l'ensemble de ses résultats possibles.

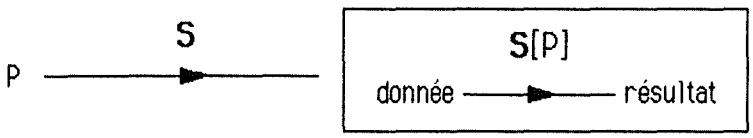

Figure 6: Sémantique associée à un programme

De manière similaire, la sémantique d'une expression expr est définie par une fonction associant, pour chaque état possible de l'environnement (par exemple, les valeurs des variables) la valeur de l'expression.

Figure 7: Sémantique associée à une expression.

Une troisième approche possible consiste à utiliser la méthode des grammaires attribuées, qui a été présentée précédemment.

On notera que ces deux dernières méthodes possèdent des caractéristiques communes:

- Toutes deux permettent de définir la sémantique des constructions composées en fonction de la sémantique de leurs composants.
- Toutes deux permettent de définir la sémantique d'une construction relativement au contexte dans lequel elle apparaît:
  - de façon explicite dans le cas de la sémantique dénotationnelle ("l'environnement"),
  - de façon implicite dans le cas de la méthode des grammaires attribuées, où le contexte peut être transmis par des attributs hérités.

Dans le cadre de ce mémoire, c'est la méthode des grammaires attribuées qui a été retenue.

6. Syntaxe concrète

Après avoir défini la syntaxe abstraite et la sémantique d'un langage, la dernière étape consiste à définir une (ou plusieurs) syntaxe(s) concrète(s).

Pour la définir, il faut d'abord choisir une représentation pour chaque construction décrite par la syntaxe abstraite. Cette représentation tiendra compte des contraintes liées au support (fins de lignes, indentations, ...), à la facilité d'utilisation (mots-clés significatifs, lisibilité du code, ...), et à la nécessité de construire un analyseur syntaxique (règles concernant l'ambiguïté, ...).

Une fois ces représentations définies, il restera à les organiser sous forme d'un "texte" constituant la description de la syntaxe concrète, que l'on pourra transmettre aux utilisateurs du langage, ainsi qu'au programmeur chargé de réaliser l'analyseur syntaxique.

Cette description peut prendre des formes diverses. Généralement, on a recours à un formalisme du type BNF, ou à une représentation sous forme de diagrammes syntaxiques.

Chapitre 3

Implémentation d'un langage

1. Introduction

Les chapitres précédents de cette troisième partie ont présenté le cadre méthodologique de la définition d'un langage de requêtes tel que PCL.

Ce chapitre présente quant à lui le cadre méthodologique de l'implémentation d'un tel langage, incluant:

- la détermination des composants du programme à implémenter,
- la présentation des outils utiles à l'implémentation de ces composants.

Dans le cadre de ce mémoire, cette implémentation devait s'effectuer en PROLOG. Tenant compte de cette contrainte, les développements de ce chapitre sont particularisés à ce langage.

2. Remarques préliminaires

On a vu que la définition d'un langage nécessite la définition de trois éléments:

- la syntaxe abstraite,
- la sémantique,
- la syntaxe concrète.

Le deux premiers éléments constituent en quelque sorte la définition "profonde", "stable" du langage. Le troisième élément, par contre, est "superficiel" et "instable", puisqu'il définit seulement la représentation des concepts du langage, et qu'il est possible d'en donner plusieurs définitions différentes.

Lors de la construction d'un programme implémentant un langage, il semble utile de tenir compte de cette distinction, afin que le programme soit:

- aisément modifiable lors de modifications de détail apportées à la syntaxe concrète,
- en grande partie réutilisable lors de la définition de plusieurs syntaxes concrètes différentes.

Evidemment, dans le cas où il s'avèrerait nécessaire d'apporter des modifications à la syntaxe abstraite et/ou à la sémantique du langage, il est clair ces modifications devraient être reportées dans l'ensemble du système.

De telles modifications sont en effet très graves, car elles portent sur les concepts même du langage. Il s'agit plus précisément

d'erreurs de spécification apparaissant à un moment très tardif de la conception du système. On peut donc espérer que de tels changements soient très rares, si pas inexistants.

3. Présentation des éléments du programme

L'implémentation d'un langage doit définir les éléments suivants:

- Une représentation interne des constructions du langage.

  En effet, la réalisation d'un programme nécessite de fixer des conventions de représentation pour les objets que celui-ci manipule.

- Différents composants constituant l'architecture du programme.

Etant donné la distinction fondamentale entre syntaxe et sémantique, il semble naturel de définir deux composants principaux:

- un composant syntaxique,
- un composant sémantique.

Dans le cas d'un langage de requêtes de type PCL, le premier composant sera chargé de vérifier la correction syntaxique des requêtes, et le second de leur évaluation. La représentation interne des expressions sera le véhicule des informations entre ces deux composants.

Les points suivants détaillent la façon dont on peut aborder en pratique ces différents éléments.

4. Représentation interne des expressions

4.1. Principe général

On a vu qu'il était utile, lors de la construction du programme, d'isoler le plus possible les aspects relatifs à la syntaxe concrète. Ce principe peut être appliqué lors du choix d'une représentation interne de la manière suivante.

Quelle que soit la représentation interne que l'on choisit, il importe de la définir dans les termes de la syntaxe abstraite, c'est-à dire qu'il faut considérer la représentation interne comme une manière de représenter des expressions abstraites. Pour ce faire, il faut soigneusement décrire les règles de cette représentation.

De cette façon, lorsque les composants du programme manipuleront la représentation interne d'expressions, ils le feront par rapport à la syntaxe abstraite.

Les points suivants décrivent deux choix possibles de représentation interne, et détaillent leurs avantages et inconvénients par rapport au principe ci-dessus.

4.2. Représentation basée sur l'expression abstraite

Au vu du principe qui vient d'être énoncé, il semble logique de définir une représentation interne calquée sur la syntaxe abstraite. Ce type de représentation doit être un "codage" systématique de la syntaxe abstraite.

Cette représentation possède les avantages suivants:

- Les règles de représentation de la syntaxe abstraite sont évidentes.
- De par les propriétés de la syntaxe abstraite, la représentation interne est débarrassée autant que possible des détails syntaxiques superflus.
- Le système est ainsi le plus indépendant possible de la syntaxe concrète. En fait, seul le composant chargé de l'analyse syntaxique devra connaître cette syntaxe concrète, pour la transformer en représentation interne.
- Ce choix facilite l'implémentation rapide d'un sous-système initial, utilisant une syntaxe concrète simplifiée, calquée sur la syntaxe abstraite.
- Un dernier avantage fondamental est que la représentation abstraite d'une expression concrète peut être enrichie d'informations que l'expression initiale ne contient pas. En effet, elle peut intégrer les résultats de l'analyse syntaxique

de cette expression (absence d'ambiguïtés, connaissance du type des sous-expressions, ...).

Ce type de représentation peut cependant présenter un inconvénient dans le cas très particulier suivant.

Si le langage défini permet de manipuler une Base de Connaissances, et que le système autorise l'utilisateur à accéder à son contenu, sans lui offrir un interface approprié, alors ce contenu doit être sous une forme directement compréhensible pour l'utilisateur.

Si de plus, dans le cadre d'un tel système, la Base de Connaissances peut contenir des objets qui constituent justement des expressions du langage de requêtes[1], alors, une représentation interne basée sur la syntaxe abstraite est inacceptable, parce qu'incompréhensible pour l'utilisateur.

On verra que cette situation s'est présentée dans le cadre de ce mémoire.

4.3. Représentation basée sur l'expression concrète

Dans le cas extrême présenté à la fin du point précédent, la seule solution possible consiste à décider que les constructions du langage seront representées dans le programme sous leur forme concrète.

Cette solution souffre évidemment des inconvénients qui faisaient l'avantage de la solution précédente.

Cependant, il reste possible d'assurer l'indépendance du programme vis-à-vis de la syntaxe concrète. Pour cela, il faut "cacher" la nature exacte de la représentation interne à l'intérieur d'un type abstrait, auquel on accède via des primitives d'accès faisant référence à la syntaxe abstraite. Il faut cependant noter que ces primitives effectueront à chaque fois une analyse syntaxique des expressions, ce qui n'est pas nécessairement efficace.

Si, finalement, cette dernière solution est également impossible à mettre en oeuvre, il reste encore possible de tirer parti des caractéristiques de stabilité de la syntaxe abstraite, en calquant la structure du programme sur celle de la syntaxe abstraite. De cette façon, la structure du programme est indépendante de la syntaxe concrète, même si ses composants ne le sont pas.

---

[1] Par exemple, des requêtes que l'utilisateur a développées, et qu'il désire pouvoir évaluer dans la suite.

5. Le composant syntaxique

Ce composant est chargé de convertir une expression concrète en sa représentation interne.

Il reçoit en entrée une chaîne de caractères censée représenter une expression concrète. Si cette chaîne représente effectivement une telle expression, il fournit en sortie sa représentation. Sinon, il renvoie un message d'erreur.

Le noyau de ce composant est constitué d'un programme appelé analyseur syntaxique, qui vérifie la correction syntaxique d'une expression concrète, et la convertit en sa représentation abstraite.

Les points suivants présentent les différentes manières de construire un analyseur syntaxique en PROLOG.

5.1. Hypothèse de travail

On a vu au chapitre précédent que la syntaxe concrète d'un langage pouvait être décrite sous des formes diverses.

Dans les développements qui vont suivre, on supposera que cette description existe sous la forme d'une grammaire BNF.

5.2. Phases de l'analyse syntaxique

Avant d'envisager la construction d'un analyseur syntaxique, il est important de préciser quelques notions permettant de distinguer les différentes phases de l'analyse syntaxique d'une expression.

Rappelons que la syntaxe concrète est un choix de représentation des constructions abstraites définies dans le langage. Dans ce contexte, deux caractéristiques fondamentales peuvent être soulignées:

- Premièrement, toute construction syntaxique concrète est une combinaison d'éléments de domaines syntaxiques concrets sur lesquels portent un certain nombre de conditions[1].

  Une construction syntaxique est donc un ensemble structuré de mots (éléments ayant un statut plus complexe que le simple caractère).

---

[1] Ces conditions reprennent notamment l'ensemble de règles de typage.

- Deuxièmement, toute construction syntaxique concrète se représente par une chaîne de caractères dans laquelle sont noyées les notions de structure et de mots. De plus, cette chaîne de caractères est généralement enregistrée sur un support, qui introduit un certain nombre de contraintes supplémentaires[1].

Ceci permet de soulever deux problèmes devant être résolus par l'analyse syntaxique :

- Etant donné une chaîne de caractères représentant une expression du langage, il s'agit de transformer cette suite de caractères en une suite de mots. Cette transformation s'effectue par la reconnaissance de mots-clés correspondant aux symboles terminaux de la grammaire.

  Cette étape est généralement appelée tokenisation, ou encore analyse lexicale.

- Etant donné une suite de mots (résultat de l'analyse lexicale), il est faut alors découvrir à quelle construction syntaxique concrète elle se rapporte, afin de retrouver la construction syntaxique abstraite correspondante.

  Cette transformation est généralement décomposée en deux phases importantes :

  - Mise en évidence de la structure sous-jacente à la suite de mots, c'est-à-dire de l'arbre syntaxique correspondant par rapport à la grammaire BNF.

    Cette phase est généralement appelée analyse syntaxique.

  - Vérification des conditions portant sur cette structure (règles de typages et autres).

    Cette phase est parfois appelée analyse sémantique, bien qu'elle n'ait rien de sémantique, puisqu'elle ne porte que sur des conditions syntaxiques de structure ou de format, qui peuvent être vérifiées une fois pour toutes "à la compilation".

    Cette confusion est probablement due au fait que, tout comme la sémantique du langage, ces conditions sont généralement exprimées en langue naturelle. Elles sont alors regroupées en une seule étape, appelée la "définition de la sémantique" du langage.

L'analyse syntaxique, au sens de la méthodologie suivie, reprend ces trois aspects de l'analyse d'une expression.

---

[1] Par exemple, les "fins de lignes", les espaces, les tabulations dans le cas où le support est un fichier.

Selon les cas, l'analyseur syntaxique correspondant pourra soit intégrer ces trois aspects en un seul programme, soit les réaliser par des programmes différents.

Chacune de ces solutions possède ses avantages et ses inconvénients. Ainsi, un analyseur où les trois aspects sont clairement distingués sera plus facile à construire, mais risquera d'être moins efficace qu'un analyseur plus complexe effectuant toutes les opérations "en une seule passe".

5.3. Techniques d'analyse syntaxique utilisables en PROLOG

Le langage PROLOG, permet d'utiliser trois grandes techniques d'implémentation d'un analyseur syntaxique.

5.3.1. Les méthodes "classiques"

Ces méthodes sont celles que l'on utilise dans les autres langages de programmation, et sont d'ailleurs indépendantes de ces langages.

Elles mettent en oeuvre un certain nombre de techniques selon le type de grammaire définissant le langage: grammaire context-free, context-sensitive, définie par un automate fini, par un automate à pile,...([LER]).

Elles permettent de construire des analyseurs syntaxiques parfaitement adaptés au problème, et de ce fait, ceux-ci peuvent être optimisés au maximum et très efficaces.

Cependant, vu l'indépendance de ces méthodes par rapport aux langages de programmation, ainsi que la généralité des concepts qu'elles manipulent, elles peuvent conduire à des programmes dont l'écriture est fastidieuse.

5.3.2. Le Prolog Reader

5.3.2.1. Description

Comme pour tout langage de programmation, les différentes implémentations de PROLOG contiennent un composant d'analyse syntaxique, qui transforme des constructions concrètes du langage (chaînes de caractères) en représentations internes.

Dans le cas de PROLOG, cette représentation interne est le terme, qui constitue également l'unique structure de données du langage. De manière simplifiée, cette structure se définit récursivement comme suit:

- les constantes et les variables sont des termes,

- une construction de la forme $f(t_1, \ldots t_n)$, où:
  - f est un identificateur appelé "foncteur",
  - $t_1, \ldots t_n$ sont des termes,
  - n est appelé l'arité du terme,

  constitue un terme.

Cette uniformité de représentation permet d'utiliser le composant d'analyse syntaxique de PROLOG (le PROLOG reader) non seulement pour analyser des programmes PROLOG, mais aussi pour analyser des constructions syntaxiques absolument quelconques, pourvu qu'elles représentent des termes.

Dès lors, une façon d'implémenter un langage en PROLOG consiste à définir sa syntaxe concrète sous forme de termes, et à utiliser directement le PROLOG reader comme analyseur syntaxique.

Pour faciliter cette utilisation, PROLOG introduit la notion d'opérateur, présentée au point suivant.

5.3.2.2. Les opérateurs PROLOG

a. Utilité

La définition d'un terme, présentée ci-dessus, conduit à une lourdeur de notation qui rend peu naturelle l'écriture de certaines constructions syntaxiques. Par exemple, on préfèrera écrire:

- P(X) & Q(Y),
- P(X,Y) <=> Q(X) & R(Y),

plutôt que:

- &(P(X) , Q(Y)),
- <=>(P(X,Y) , &(Q(X) , R(Y))).

A cet effet, le langage PROLOG permet d'assouplir la notation de certains foncteurs, en remplaçant la notation "fonctionnelle" des termes correspondants par une notation infixée, préfixée ou postfixée, dans laquelle les parenthèses peuvent être omises.

Pour cela, il suffit de déclarer ces foncteurs comme des opérateurs. Cette déclaration est différente en PROLOG selon que les foncteurs sont d'arité un et deux, ou d'arité supérieure.

b. Opérateurs d'arité un et deux

b.1. Caractéristiques générales

Un tel opérateur est caractérisé par les notions suivantes:

- son **nom**,
- son **arité**,
- sa **position** par rapport à ses opérandes,
- son **niveau de priorité**,
- son **associativité**.

Les deux premières notions découlent de la notion de foncteur. Les trois autres sont définies comme suit.

La **position** de l'opérateur par rapport à ses opérandes

- La position préfixée indique que l'opérateur se place avant les opérandes:

  ex.: l'opérateur de négation logique: ¬A.

- La position postfixée indique que l'opérateur se place après les opérandes:

  ex.: l'opérateur "factorielle" : n! .

- La position infixée indique que l'opérateur (d'arité deux uniquement) se place entre les opérandes:

  ex.: l'opérateur d'addition : x + y .

Le niveau de **priorité** de l'opérateur

Pour l'ensemble des opérateurs, il est nécessaire de définir une relation d'ordre, définissant un graphe de niveaux de priorité.

On représentera cette notion de priorité en numérotant les différents niveaux et en associant à chaque opérateur le numéro de son niveau dans le graphe. Les niveaux sont numérotés en ordre décroissant, la plus grande priorité recevant le numéro le plus petit.

ex.: l'expression:

x + y * z,

où le niveau de priorité de + est plus grand que celui de *, est équivalente à:

x + (y * z).

L'**associativité**

Cette propriété définit la façon dont il faut régler le conflit de priorité qui se présente lorsque dans une expression se retrouvent deux opérateurs ayant des niveaux de priorité identiques.

Autrement dit, l'associativité permet de déterminer si l'expression doit être évaluée de gauche à droite ou de droite à gauche.

ex.: dans le cas de l'expression x / y * z, où / et * ont même niveau de priorité, comment faut-il évaluer cette expression ? x / (y * z), ou (x / y) * z ?

La manière dont il faut régler les conflits de priorité pour un opérateur est déterminée par les deux points suivants:

1. Pour tout opérande doit être précisé un niveau de priorité, défini par les règles suivantes:

   "Toute expression primitive est de priorité 0."

   "Toute expression entre parenthèses est de priorité 0."

   "La priorité d'une expression contenant des opérateurs est celle de son opérateur le moins prioritaire."

2. Il faut également préciser, pour chaque opérande, si son niveau de priorité doit être:

   - strictement inférieur, ou
   - inférieur ou égal

   à la priorité de l'opérateur.

Ces définitions permettent de déterminer la manière dont il faut regrouper les opérandes autour des opérateurs en l'absence de parenthèses.

b.2. Définition Prolog

Les opérateurs d'arité un et deux sont définis en PROLOG par une expression de la forme:

OP ("entier", "règle", "nom" **).**

La signification de cette expression est la suivante ([CLOCK-MELL]):

- "nom" est le nom de l'opérateur.
- "entier" définit son niveau de priorité. Il est généralement compris dans l'intervalle [0, 255] ou dans l'intervalle [1, 1200] selon l'implémentation du langage PROLOG.
- "règle" définit à la fois l'arité, la position, et l'associativité de l'opérateur, au moyen d'une combinaison de trois symboles:
  - La position du symbole "f" représente la position de l'opérateur, et celle des symboles "x" ou "y" représente la position des opérandes.
  - Le symbole "y" désigne un opérande dont le niveau de priorité peut égaler celui de l'opérateur, tandis que le symbole "x" désigne un opérande dont le niveau de priorité doit être strictement inférieur à celui de l'opérateur.

  Une telle règle peut prendre l'une des formes: xfx, xfy, yfx, xf, yf, fx, fy.

A titre d'exemple:

Dans le cas de BIM-Prolog, l'intervalle de priorité des opérateurs est [1, 1200]. Les opérateurs arithmétiques sont définis de la façon suivante:

- addition: OP(500, yfx, +),
- soustraction: OP(500, yfx, -),
- multiplication: OP(400, yfx, *)
- division: OP(400, yfx, /)

Ces définitions correspondent aux conventions habituelles, où la multiplication et la division sont plus prioritaires que l'addition et la soustraction, et où les opérations de même niveau sont effectuées de gauche à droite.

c. Opérateurs d'arité supérieure à deux

Le langage PROLOG ne permet pas de définir explicitement des opérateurs d'arité supérieure à deux. On peut cependant le faire en recourant à quelques artifices.

Ce point montre comment on peut traiter les opérateurs d'arité 3. La définition d'opérateurs d'arité supérieure se base sur les même principes.

Un opérateur d'arité 3 est de la forme:

OP1 opérande1 **OP2** opérande2 **OP3** opérande3

où OP1, OP2, et OP3 sont trois mots-clé [1].

Les notions générales caractérisant un tel opérateur sont une restriction de celles caractérisant tout opérateur d'arité 1 ou 2:

- Son **nom** (composé de ses trois mots-clé)
- Son **arité**: le nombre de ses opérandes
- Son **niveau de priorité**

Les notions telles que la position par rapport aux opérandes ou les règles d'associativité n'ont pas de sens dans ce contexte.

La manière de définir un tel opérateur en PROLOG est de le considérer comme une combinaison d'opérateurs d'arité 1 et 2, qui respectent des règles d'associativité judicieusement choisies.

Ces règles doivent permettre à l'utilisateur de manipuler sans ambiguïté et sans devoir utiliser de parenthèses des expressions de la forme:

OP1 opérande1 **OP2** opérande2 **OP3** opérande3

qui seront toujours interprétées, par exemple, comme:

OP1 ((opérande1 **OP2** opérande2) **OP3** opérande3)

Pour cela, tout opérateur concret d'arité 3 peut être défini au moyen de trois opérateurs PROLOG:

- Le premier opérateur PROLOG défini est d'arité 1.

  Sa définition est la suivante:

  nom: OP1

  priorité: niveau de priorité de l'opérateur concret

  règles d'associativité: fx

---

[1] Par exemple, dans le cas de PCL,

EXIST variable **MEMBER** setofvalues **WITH** condition

Ou plus simplement, dans la plupart des langages de programation classique,

IF condition **THEN** instruction_1 **ELSE** instruction_2

- Le troisième opérateur est un opérateur d'arité 2.

  Sa définition est la suivante:

  nom: OP3

  priorité: strictement inférieure à celle de OP1

  règles d'associativité: xfy

- Le deuxième opérateur est un opérateur d'arité 2.

  Sa définition est la suivante:

  nom: OP2

  priorité: strictement inférieure à celle de OP3

  règles d'associativité: xfx

Il faut également tenir compte de règles syntaxiques supplémentaires, vérifiant que:

- L'opérande de OP1 est toujours de la forme:

  $expr_{int}$ **OP3** opérande3

- L'expression $expr_{int}$ est toujours de la forme:

  opérande1 **OP2** opérande2

- Les expressions opérande1, opérande2, opérande3 sont des termes PROLOG syntaxiquement corrects.

5.3.2.3. Grammaires adaptées au PROLOG reader

Les points précédents ont montré comment on pouvait tirer parti du PROLOG reader pour analyser un langage "décrit en termes d'opérateurs". Il reste à montrer comment un langage peut être effectivement "décrit en termes d'opérateurs".

Cette description se ramène en fait en une description de type BNF, dans laquelle les constructions du langage sont définies selon une hiérarchie de niveaux. Dans cette hiérarchie:

- le niveau zéro regroupe la définition des expressions "de base" du langage (constantes, identificateurs, ...). On considère en outre qu'une expression de niveau quelconque, placée entre parenthèses, est une expression du niveau zéro.

- les autres niveaux définissent des expressions constituées d'un opérateur et de ses opérandes. Ces opérandes sont soit des expressions de niveau inférieur, soit des expressions de même niveau (dans le cas de règles récursives).

Une telle grammaire peut donc contenir des règles similaires aux règles suivantes (les niveaux sont numérotés de 0 à N):

- Règles de niveau 0:

  ‹Expr$_0$› ::= ‹Constante› | ‹Identificateur› | ... | (‹Expr›)

- Règles de niveau i:

  ‹Expr$_i$› ::= ‹Expr$_{i-1}$› | ‹Construction›,

  où ‹Construction› peut être définie au moyen d'une des règles suivantes:

  ‹Construction› ::= ‹Expr$_{i-1}$› ‹OP1$_i$› ‹Expr$_{i-1}$›

  ‹Construction› ::= ‹Expr$_{i-1}$› ‹OP2$_i$› ‹Expr$_i$›

  ‹Construction› ::= ‹Expr$_i$› ‹OP3$_i$› ‹Expr$_{i-1}$›

  ‹Construction› ::= ‹Expr$_{i-1}$› ‹OP4$_i$›

  ‹Construction› ::= ‹Expr$_i$› ‹OP5$_i$›

  ‹Construction› ::= ‹OP6$_i$› ‹Expr$_{i-1}$›

  ‹Construction› ::= ‹OP7$_i$› ‹Expr$_i$›

- Règle principale:

  ‹Expr› ::= ‹Expr$_N$›

Sur base d'une telle grammaire, il est aisé de définir les opérateurs PROLOG correspondants:

- Le niveau de priorité d'un opérateur est défini à partir du niveau de l'expression dans laquelle il apparaît,

- Ses règles d'associativité sont déduites de sa position dans la règle grammaticale correspondante, et du niveau de priorité de ses opérandes.

  Ainsi, dans les exemples de règles ci-dessus, les opérateurs OP1$_i$, OP2$_i$, ... OP7$_i$ possèdent respectivement les règles d'associativité : xfx, xfy, yfx, xf, yf, fx et fy.

On peut remarquer que l'ensemble des grammaires pouvant être définies de cette manière est une restriction de l'ensemble des grammaires BNF.

En effet, il est quelquefois possible de transformer une grammaire BNF en une telle grammaire. Dans un premier temps, il faut hiérarchiser ses règles suivant un certain nombre de niveaux. Ensuite, il faut choisir judicieusement les symboles terminaux qui correspondront à des opérateurs.

Il est clair cependant que cette transformation ne pourra pas s'opérer facilement dans tous les cas. Certaines grammaires peuvent par exemple posséder de nombreuses règles mutuellement récursives, qui ne facilitent pas la hiérarchisation. D'autres peuvent pos-

séder trop d'opérateurs au même niveau, pouvant donner lieu à des problèmes d'ambiguïté. D'autres enfin peuvent utiliser un même symbole terminal dans différents contextes, ce qui exigerait par exemple la définition d'opérateurs de même nom, mais d'arités différentes; or, une telle définition est impossible en PROLOG.

5.3.2.4. Avantages

Les avantages d'une utilisation du PROLOG reader sont les suivants:

- Rapidité d'implémentation, dans le cas où la grammaire s'y prête.
- Méthode particulièrement adaptée dans le cas où le langage comporte peu de règles syntaxiques supplémentaires à la définition des opérateurs.
- Il n'est pas nécessaire d'implémenter explicitement la phase de tokenisation.
- Un changement de notation d'un opérateur n'affecte que la définition de cet opérateur.

5.3.2.5. Inconvénients

Les inconvénients d'une utilisation du PROLOG reader sont les suivants:

- Le contrôle des erreurs durant la phase de tokenisation échappe totalement au programmeur: toute erreur à la lecture d'une chaîne de caractères par le PROLOG reader a pour effet de renvoyer un message d'erreur à l'écran.
- Il est impossible d'optimiser la phase de tokenisation.
- Seuls les opérateurs d'arité 1 ou 2 sont reconnus par le PROLOG reader. Pour définir d'autres opérateurs, il faut recourir à un artifice de programmation.
- Pour être reconnus, les opérateurs doivent être distincts des opérandes dans la chaîne d'entrée (séparés par des blancs ou commençant par des caractères spéciaux).
- Il est impossible de traiter avec cette méthode des langages dans lesquels un même nom peut être donné à deux opérateurs utilisés dans des contextes distincts.

En conclusion, cette méthode s'applique uniquement dans le cas de grammaires simples, context-free et définissables en termes d'opérateurs univoques.

5.3.3. Les Definite Clause Grammars

5.3.3.1. Description

Les DCG's (Definite Clause Grammars) sont une extension des grammaires "context-free", introduites pour réaliser facilement des analyseurs syntaxiques en PROLOG. A cet effet, un formalisme propre (du type BNF) est proposé.

De plus, le langage PROLOG incorpore généralement un composant qui génère, à partir d'une telle grammaire, un programme PROLOG constituant l'analyseur syntaxique du langage[1].

Cette extension des grammaires "context-free" est réalisée de deux manières :

- Les DCG's permettent d'adjoindre des paramètres aux règles grammaticales.

  Ces paramètres permettent de transmettre des informations d'une règle grammaticale vers les règles définissant ses composants (mécanisme similaire à celui des attributs hérités), et vice-versa (mécanisme similaire à celui des attributs synthétisés).

- Les DCG's permettent également d'associer à une règle grammaticale une suite de buts PROLOG, qui seront exécutés lorsque l'analyseur syntaxique appliquera cette règle.

  Ces buts PROLOG peuvent bien sûr manipuler les paramètres transmis entre les règles.

  Ils permettent par exemple de vérifier des conditions syntaxiques supplémentaires (typage, ...), ainsi que de construire une représentation interne des expressions analysées.

Cette technique est beaucoup plus puissante que celle du Prolog reader. Elle est aussi applicable dans la plupart des cas.

Cependant, la traduction des règles DCG's conduit à des analyseurs syntaxiques récursifs utilisant le mécanisme de backtracking de PROLOG de manière intensive, ce qui peut nuire à leur efficacité.

Malgré tout, elle permet la définition de langages ne pouvant être décrits par des grammaires context-free.

---

[1] Remarquons que si le langage Prolog ne contient pas un tel composant, il est toujours possible de l'implémenter à partir de l'algorithme décrit dans [CLOCK-MELL].

5.3.3.2. Avantages

Les avantages d'une définition de la grammaire dans le formalisme des DCG's sont les suivants :

- L'implémentation de l'analyseur pour une grammaire BNF est très facile, puisque le formalisme utilisé est le même.
- Les modifications de la grammaire sont automatiquement reportées sur l'analyseur, par le programme de conversion.
- Le programme écrit est très clair, puisque son texte est constitué de celui de la grammaire.
- Cette méthode facilite le prototypage rapide de grammaires.
- Elle est également plus puissante que l'utilisation du Prolog reader.

5.3.3.3. Inconvénients

L'utilisation d'une grammaire DCG présente un inconvénient lié à la différence entre la nature déclarative de ce formalisme et la nature procédurale du processus d'analyse syntaxique:

- si la grammaire est écrite de façon "lisible" et "naturelle" pour un être humain, elle peut résulter en un programme très inefficace,
- par contre, si on "optimise" l'écriture d'une grammaire afin d'obtenir un programme très efficace, elle se met à ressembler de plus en plus à un programme PROLOG "farci" d'artifices de programmation.

Cet inconvénient est dû à ce que le formalisme DCG a été conçu comme une simple "amélioration syntaxique" de PROLOG. De ce fait, la traduction d'une grammaire DCG en un programme PROLOG est effectuée de façon rudimentaire, sans la moindre tentative d'optimisation du code obtenu[1].

Un second inconvénient réside dans le fait que, généralement, le composant DCG incorporé à PROLOG exige que la chaîne d'entrée se présente sous forme d'une liste.

Une lecture préalable est donc obligatoire si la chaîne se trouve initialement sur fichier. De plus, cette lecture charge la chaîne complètement en mémoire centrale, ce qui peut poser des problèmes si elle est très grande (par exemple, dans le cas d'un compilateur).

---

[1] On peut d'ailleurs constater ce fait en consultant l'algorithme proposé dans [CLOCK-MELL].

6. Le composant sémantique

Dans le cas d'un langage de requêtes, ce composant se ramène à un programme chargé de l'évaluation d'une requête.

Ce programme reçoit en entrée la représentation interne d'une expression syntaxiquement correcte. Il fournit en sortie le résultat de son évaluation.

Il faut noter que dans le cas où l'on a choisi comme représentation interne des expressions leur forme syntaxique concrète, l'écriture de ce programme peut être compliquée.

En effet, si la représentation concrète n'a pas été isolée dans un type abstrait convertissant cette représentation en une structure syntaxique abstraite, le composant sémantique devra effectuer lui-même cette conversion, qui, on l'a vu, se ramène à une analyse syntaxique.

Quatrième partie

Le langage PCL

Après avoir défini les objectifs du mémoire, et précisé le cadre informatique dans lequel il se situe, on a présenté dans la troisième partie la base théorique sur laquelle reposent la définition et l'implémentation du langage PCL.

Cette quatrième partie présente cette définition et cette implémentation. Sa structure est similaire à celle de la troisième partie. Elle présente la manière dont les théories et méthodes traitées dans cette dernière ont été mises en pratique dans le cas de PCL. Ainsi, on décrira le méta-modèle de Bases de Connaissances de PCL, la définition de ce langage, et son implémentation.

Comme on l'a dit dans l'introduction générale de ce mémoire, cette partie constitue un résumé de la démarche suivie, qui est détaillée complètement dans les annexes 1 à 11.

Chapitre 1

Définition du méta-modèle de Bases de Connaissances de PCL

1. Introduction

Comme il a été présenté au chapitre 1 de la troisième partie, une étape préliminaire de la définition d'un langage de requêtes consiste à décrire son méta-modèle conceptuel de Base de Connaissances.

La méthode suivie dans le cas du langage PCL décrit le modèle conceptuel, ainsi que la sémantique qui lui est associée, selon une structure hiérarchique de niveaux de cohérence.

La définition complète du méta-modèle conceptuel et de sa sémantique constitue l'annexe 1 ("Méta-modèle conceptuel, définition"). Cette définition a pû être réalisée sur base des articles présentant les langages PROBE, et TDL ([CCI-BIM-UF], [MEIR-TR-VEN], [MEIR-TR]).

L'annexe 2 ("Schéma des niveaux de cohérence du modèle conceptuel") présente un schéma de la structure hiérarchique de la définition, ainsi qu'une liste détaillée des concepts définis à chaque niveau.

Ce chapitre présente la hiérarchie choisie, en justifiant les choix qui ont été effectués lors de son élaboration.

2. Présentation de la hiérarchie

La hiérarchie de niveaux de cohérence a été définie en se basant sur les réflexions suivantes.

De manière intuitive, le contenu d'une Base de Connaissances est constitué de classes et d'instances, pouvant être reliées à d'autres par des relations. Pour cela, un méta-modèle conceptuel doit définir les notions de classes, d'instances, de slots, de relations,...

Or, dans le cas de PCL, l'un des types de relations possibles (relations entre classes) est défini au moyen d'expressions PCL. Cette caractéristique peut donc conduire à des définitions circulaires:

- les relations sont exprimées en termes d'expressions PCL,
- les expressions PCL sont définies en termes du contenu d'une Base de Connaissances, contenant notamment des relations.

Pour répondre à ce problème, on peut proposer une première hiérarchie de niveaux:

- Un premier niveau définit de manière très générale les objets de base du méta-modèle (classes, instances, slots, facettes, relations...), sans introduire aucune condition restrictive.

- Un second niveau raffine la notion de facette, car lorsque l'on définira une relation au moyen d'une expression PCL, on le fera via une facette.

- Un troisième niveau définit les expressions PCL, sur base des niveaux inférieurs.

Cette première hiérarchie doit cependant être raffinée. D'abord, il s'agit de distinguer la définition des facettes et les conditions de leur utilisation. Ensuite, il faut distinguer la définition générale d'une expression PCL, et les restrictions à lui apporter dans certains cas particuliers. Les niveaux deux et trois ci-dessus doivent donc être divisés en deux. On obtient ainsi la hiérarchie suivante:

- Définitions générales des "objets de base" du modèle (niveau de cohérence ZERO).

- Définition de l'objet "définition de facette" (niveau de cohérence UN).

- Conditions et propriétés portant sur les objets du modèle (niveau de cohérence DEUX).

- Définition d'une expression PCL générale (niveau de cohérence TROIS).

- Restrictions sur l'utilisation des expressions PCL dans les cas particuliers de définitions de slots et requêtes (niveau de cohérence QUATRE).

Les points suivants détaillent, pour chacun de ces niveaux, les définitions syntaxiques et sémantiques auxquelles il donne lieu.

3. Niveau de cohérence ZERO

Ce niveau regroupe donc les définitions générales des "objets de base" du modèle. Sa raison d'être est d'établir la liste des objets qui seront raffinés dans les niveaux supérieurs.

Du point de vue syntaxique, il énonce des propriétés très générales concernant les objets (par exemple, "une relation IS-A est un couple de classes"). Il en donne également une première définition sémantique.

4. Niveau de cohérence UN

Ce niveau raffine la définition de l'objet "définition de facette".

Du point de vue syntaxique, le niveau de cohérence ZERO avait défini une définition de facette comme un "couple de valeurs". Ce niveau précise pour chaque valeur possible du premier argument la forme du second. Il en donne une définition sémantique, excepté

dans les cas où cette définition nécessite la connaissance de la sémantique d'une expression PCL.

5. Niveau de cohérence DEUX

On a vu que l'existence de ce niveau était requise afin de préciser les conditions dans lesquelles les différents types de facettes étaient utilisables.

Par la même occasion, on définit également à ce niveau l'ensemble des conditions et propriétés portant sur les autre objets du modèle (par exemple, les propriétés des relations IS-A). Notons cependant que certaines conditions et propriétés, faisant référence aux expressions PCL, ne pourront êtres définies à ce niveau.

Ces différentes conditions font l'objet d'une définition syntaxique et sémantique.

6. Niveau de cohérence TROIS

Ce niveau couvre principalement les aspects syntaxiques et sémantiques de la définition d'une expression PCL. Il contient également les conditions et propriétés faisant référence à la définition d'une telle expression.

La cohabitation de ces deux types de définitions n'introduit pas de circularité. En effet, si la définition des conditions et propriétés fait référence à celle d'une expression, cette dernière ne fait référence qu'à des notions définies aux niveaux inférieurs.

De plus, le regroupement de ces deux types de définitions en un seul niveau donne une meilleure cohésion au méta-modèle. On peut en effet considérer que ce modèle est complet au niveau TROIS: le niveau supérieur n'introduit qu'un certain nombre de restrictions concernant l'utilisation des expressions dans la définition d'un modèle.

7. Niveau de cohérence QUATRE

Ce niveau définit les restrictions qu'il faut apporter aux expressions PCL lorsqu'on les utilise dans certains cas particuliers.

Par exemple, lorsque l'on définit un slot au moyen d'une expression PCL, on ne peut utiliser qu'une classe restreinte d'expressions.

Chapitre 2

Définition du langage PCL

1. Introduction

Ce chapitre montre comment on a défini le langage PCL suivant la méthode décrite au chapitre 2 de la troisième partie. Il présente successivement:

- l'identification des concepts de PCL,
- sa syntaxe abstraite,
- sa sémantique,
- sa syntaxe concrète.

2. Identification des concepts de PCL

Sur base du contenu attendu de PCL, décrit dans la première partie ([MEIR-TR], [MEIR-TR-VEN]), on a procédé à une classification des expressions du langage, de la manière suivante:

- Les expressions primitives:
  - les constantes (entiers, réels, strings),
  - les variables (pouvant remplacer une expression PCL de type quelconque),
  - les noms de classes (désignant des classes définies dans le modèle conceptuel),
  - les noms de slots (désignant des slots définis dans le modèle conceptuel).
- Les expressions composées:
  - les ensembles explicites de valeurs (énumérations de constantes de même type),
  - les tuples (énumération d'expressions PCL de type quelconque),
  - les expressions contenant un opérateur. On distingue les opérateurs:
    - ensemblistes ($\subset$, $\in$)
    - logiques ($\wedge$, $\vee$, $\neg$, $\forall$, $\exists$)
    - arithmétiques (+, -, *, /)
    - de comparaison ($<$, $\leq$, =, $\geq$, $>$, $\neq$)
    - de calcul sur des ensembles,
    - désignant la classe associée à un slot,
    - définissant un ensemble,
    - posant une condition sur un objet.

La description intuitive de ces expressions est reprise à l'annexe 4 ("Présentation intuitive des expressions du langage"). Cette annexe donne en outre, pour chaque expression, l'énoncé de sa syntaxe abstraite, ainsi qu'une description intuitive de sa sémantique.

3. Syntaxe abstraite de PCL

Les points suivants présentent la syntaxe abstraite de PCL, et décrivent la méthode utilisée pour vérifier la correction syntaxique des expressions.

3.1. Définition de la syntaxe abstraite

Cette syntaxe a été définie sur base de la classification des expressions PCL.

3.1.1. Expressions primitives

Ces expressions sont représentées de la manière suivante:

- Les variables et les constantes sont représentées de manière classique.
- Les noms de classes et les noms de slots étant des objets du modèle conceptuel, aucun choix de représentation n'a dû être effectué.

En outre, le type de toute expression primitive est également défini.

On trouvera ces définitions dans les annexes 4 ("Présentation intuitive des expressions du langage") et 5 ("Grammaire attribuée définissant la syntaxe abstraite").

3.1.2. Expressions composées

Les ensembles explicites de valeurs et les tuples sont représentés de la façon suivante:

- Un ensemble explicite de valeurs est représenté par une liste de constantes inscrites entre crochets.
- Un tuple est représenté par une liste d'expressions PCL, inscrites entre parenthèses.

En outre, le type de ces expressions est également défini.

Les expressions contenant des opérateurs sont définies de la façon suivante:

- elles sont représentées à la manière des termes PROLOG, où le foncteur représente l'opérateur, et les arguments représentent les opérandes.

- pour chaque opérateur, une table définit le type de ses arguments, le type de l'expression, et les règles de compatibilité de types.

Ces différentes définitions se trouvent dans l'annexe 5 ("Grammaire attribuée définissant la syntaxe abstraite").

3.2. Règles de correction syntaxique

Ces règles ont été définies au moyen d'une grammaire attribuée, dont la description complète se trouve à l'annexe 5 ("Grammaire attribuée définissant la syntaxe abstraite"). Notons que la correction syntaxique d'une expression est toujours définie par rapport à un contexte, constitué notamment d'un modèle conceptuel. Ce contexte est précisé en annexe 3 ("Contexte de définition d'une expression").

Les points suivants décrivent chacun des attributs qui ont été définis.

3.2.1. L'attribut "synt"

Cet attribut définit la correction syntaxique des expressions, et constitue l'attribut principal de la grammaire. Il associe à chaque expression une des valeurs { correct , not correct }, sur base des règles suivantes:

- Correction d'une expression primitive:
  - la correction syntaxique d'une constante ne pose pas de problème particulier,
  - une variable est correcte si elle a été déclarée (par l'emploi d'un opérateur adéquat),
  - les noms de classes et de slots sont corrects s'ils sont définis dans le modèle conceptuel considéré.
- Correction d'une expression composée:
  - un ensemble explicite de valeurs est correct si chacune de ses valeurs est correcte,
  - un tuple est correct si chacune des expressions qu'il contient est correcte,
  - une expression contenant un opérateur est correcte si:
    - chacune de ses sous-expressions est correcte
    - les règles de concordance de types définies pour l'opérateur sont vérifiées.

On constate à la lecture de ces règles que l'attribut est synthétisé pour les expressions composées à partir de leurs composants. La mise en oeuvre de cet attribut nécessite la définition d'un certain nombre d'attributs auxiliaires, définis aux points suivants.

3.2.2. L'attribut "type"

Ce premier attribut auxiliaire permet de vérifier les règles de concordance de types. Il associe à toute expression le type qui lui correspond, selon les règles définies par la syntaxe abstraite.

On peut remarquer que cet attribut est synthétisé pour la plupart des expressions, mais qu'il est hérité dans le cas des variables (à partir de leurs déclarations).

3.2.3. L'attribut "object"

Cet attribut auxiliaire est défini dans le but de faciliter la classification des objets manipulés par les règles. Il associe à chaque expression l'une des valeurs suivantes:

- cst dans le cas d'une expression constante
- class dans le cas d'un nom de classe
- var dans le cas d'une variable
- cond dans le cas d'une expression booléenne
- exp dans les autres cas.

3.2.4. Les attributs "inh-ref" et "ref"

L'ensemble des classes qui sont référencées dans une expression intervient dans la détermination de sa correction syntaxique (voir l'annexe 3: "Contexte pour la définition d'une expression").

En effet, lorsque l'on analyse une expression qui est sous-expression d'une autre, il est nécessaire de connaître l'ensemble des classes qui ont déjà été référencées. Cet ensemble est transmis aux sous-expressions par l'attribut hérité "inh-ref".

D'autre part, lorsque l'on a analysé une sous-expression, il est utile de disposer de l'ensemble des classes qui y ont été référencées, afin de l'ajouter à l'ensemble des classes déjà référencées, et de le transmettre aux sous-expressions suivantes de l'expression principale. Cet ensemble est représenté par l'attribut synthétisé "ref".

Ces deux attributs associent donc à chaque construction deux listes de classes. Ces listes sont ordonnées selon le principe "Last in -

First out". Cet ordre permet, lorsque l'on référence un slot sans préciser la classe à laquelle il appartient, de déterminer sans ambiguïté de quelle classe il s'agit, en prenant dans la liste la première classe pour laquelle le slot est défini.

3.2.5. Les attributs "inh-tabVar" et "tabVar"

De manière similaire aux attributs "inh-ref" et "ref", on définit les attributs "inh-tabVar" et "tabVar", dont le rôle est de véhiculer, lors de l'analyse d'une expression, l'ensemble des variables qui y sont déclarées.

Ces attributs associent à chaque expression une liste dont l'ordre n'a pas d'importance, et qui contient, pour chaque variable, son nom, son type et la classe qu'elle référence.

3.2.6. L'attribut "class-refer"

Ce dernier attribut facilite la détermination de la classe à laquelle une expression donnée fait référence. Il est notamment utilisé pour simplifier la gestion des listes de classes et de variables, fournies par les attributs "inh-ref", "ref", "inh-tabVar" et "tabVar".

4. Sémantique de PCL

La sémantique de PCL a été définie au moyen d'une grammaire attribuée, dont la description complète se trouve à l'annexe 7 ("Grammaire attribuée définissant la sémantique"). Comme dans le cas de la syntaxe, la sémantique d'une expression est toujours définie par rapport à un contexte, constitué notamment d'un modèle conceptuel, précisé en annexe 3 ("Contexte de définition d'une expression").

Les points suivants décrivent chacun des attributs qui ont été définis.

4.1. L'attribut "eval"

Cet attribut définit la sémantique des expressions, et constitue l'attribut principal de la grammaire. Il associe à toute expression:

- Soit un ensemble, dont chaque élément est une valeur possible de l'expression pour un jeu d'instanciations particulier des variables et classes qui y apparaissent.
- Soit la valeur error, dans le cas où l'expression n'a pas de sens.

Selon son type, la valeur d'une expression peut être:

- une instance,
- un ensemble d'instances,
- un tuple,
- un booléen.

Cet attribut est synthétisé, puisque la valeur d'une expression est calculée à partir de ses composants. Sa mise en oeuvre nécessite la définition d'un certain nombre d'attributs auxiliaires, décrits aux points suivants.

4.2. L'attribut "denote"

Certaines expressions PCL permettent de décrire des ensembles. Ces descriptions sont de deux types:

- L'expression "dénote" l'ensemble. Son évaluation produira un ensemble de valeurs dont les éléments seront utilisés dans la suite.
- L'expression définit l'ensemble, à la façon d'une définition de type. L'appartenance d'un élément à cet ensemble est alors

vue comme une déclaration de l'appartenance de l'élément à un type restreint. Ce type d'expression peut être utilisé dans une autre pour définir le type des variables qui y apparaissent.

Il est donc nécessaire de pouvoir distinguer si une expression doit être considérée comme une expression à évaluer ou comme une déclaration de type. Ce rôle est joué par l'attribut "denote".

4.3. L'attribut "insForEval"

Une des requêtes les plus simples exprimables dans le langage PCL consiste à obtenir la valeur d'un slot d'une instance de classe.

Par exemple, on dispose d'une Base de Connaissances contenant une définition de la classe "Livre" pour laquelle est défini un slot indiquant l'année d'édition. Une telle requête consiste à demander l'année d'édition pour une instance déterminée de la classe "Livre".

Cette requête fait apparaître la nécessité de pouvoir, dans certains cas, préciser pour quelle instance particulière de la Base de Connaissances l'expression doit être évaluée. Ce rôle est joué par l'attribut "insForEval".

4.4. Les attributs "inh-Table" et "Table"

Ces attributs jouent un rôle similaire aux attributs "inh-ref" et "ref", ainsi qu'aux attributs "inh-tabVar" et "tabVar" de la grammaire attribuée définissant la correction syntaxique.

Il permettent de véhiculer, lors de l'évaluation d'une expression, l'ensemble des variables et des classes déjà rencontrées, ainsi que l'instanciation courante de chacune d'elles.

La mise en oeuvre de ces attributs nécessite la définition d'un attribut auxiliaire "inh-ins". Son rôle, très technique et complexe, est précisé en annexe 7.

5. Syntaxe concrète de PCL

La définition de cette syntaxe concrète se base sur les travaux préliminaires de [MEIR-TR-VEN] et [MEIR-TR]. Une contrainte importante sur cette définition était que l'analyseur syntaxique (que l'on décrira au chapitre suivant) devait être implémenté en utilisant la technique du PROLOG reader (décrite au chapitre 3 de la troisième partie). Pour cela, la syntaxe était pensée dès le début en termes d'opérateurs.

Cette syntaxe est constituée d'une grammaire BNF, et de règles syntaxiques supplémentaires, qui n'ont pas pu être exprimées dans ce formalisme.

5.1. Grammaire BNF

La grammaire est présentée ci-dessous dans le formalisme BNF habituel. On a cependant introduit une notation permettant de simplifier la définition de listes d'éléments. La règle:

<élément 1> <élément 2>$^{1..\infty}$ <élément 3>

est équivalente à :

<construction> ::= <élément 1> <élément 2> <suite>

<suite> ::= <élément 3> | , <élément 2> <suite>

Règles grammaticales:

<expression> ::= <primitive> | <composée>

<primitive> ::= <constante> | <variable> | <classe> | <slot>

<composée> ::= <ensemble> | <tuple> | <opérateur>

<constante> ::= *constante PROLOG*

<variable> ::= *atome PROLOG*

<classe> ::= *atome PROLOG*

<slot> ::= *atome PROLOG*

<ensemble> ::= **[** <constante>$^{1..\infty}$ **]**

<tuple> ::= **(** <expression>$^{1..\infty}$ **)**

<opérateur> ::= <disjonction>

<disjonction> ::= <conjonction> **OR** <disjonction> | <conjonction>

<conjonction> ::= <négation> **AND** <conjonction> | <négation>

<négation> ::= **NOT** <négation> | <expression arithmétique>

<expression arithmétique> ::=
<expression arithmétique> **PLUS** <terme> |
<expression arithmétique> **MINUS** <terme> |
<terme>

<terme> ::= <terme> **TIMES** <fonction> | <terme> **DIV** <fonction> |
<fonction>

<fonction> ::= **SETOF** <fonction> | **AVG** <fonction> | **MIN** <fonction> |
MAX <fonction> | **SUM** <fonction> | **COUNT** <fonction> |
<opérateur ternaire> | <restriction>

<restriction> ::= <relation> **WHERE** <restriction> | <relation>

<relation> ::= <objet> **EQ** <objet> | <objet> **NE** <objet> |
<objet> **GT** <objet> | <objet> **GE** <objet> |
<objet> **ST** <objet> | <objet> **SE** <objet> |
<objet> **ISIN** <objet> | <objet> **SETEQ** <objet> |
<objet> **MEMBER** <objet> |
<objet> **INCLUDED** <objet> | <objet>

<objet> ::= <définition de variable> **#** <objet> | <définition de variable>

<définition de variable> ::= **?** <variable> | <expression de base>

<expression de base> ::= <primitive> | <ensemble> | <tuple> | **(** <expression> **)**

<opérateur ternaire> ::= <mot 1> <objet> <mot 2> <objet> <mot 3> <restriction>

<mot 1> ::= **EXIST** | **FORALL**

<mot2> ::= **MEMBER** | **INCLUDED**

<mot3> ::= **WITH**

5.2. Conditions syntaxiques supplémentaires

La grammaire BNF présentée ci-dessus ne suffit pas à définir la syntaxe de PCL. Il faut en effet tenir compte de règles supplémentaires.

D'abord, les sous-expressions intervenant dans des expressions contenant un opérateur (**AND**, **PLUS**, **SETOF**, ...) doivent satisfaire des règles de typage. Ces règles sont exprimées au moyen de tables de concordance de types, reprises à l'annexe 8 ("Choix d'une syntaxe concrète").

Ensuite, un certain nombre de conditions syntaxiques plus complexes doivent être exprimées. Elles concernent par exemple la déclaration des variables utilisées dans les expressions, ou les contraintes sur les mots-clés constituant des "opérateurs ternaires". Ces conditions sont définies par la grammaire attribuée définissant la correction syntaxique des expressions (annexe 5). Elles sont présentées de manière informelle dans l'annexe 8.

Chapitre 3

Implémentation de PCL

1. Introduction

Ce dernier chapitre présente la manière dont on a implémenté le langage PCL, par rapport aux définitions théoriques du chapitre 3 de la troisième partie. Il explique successivement les choix effectués du point de vue:

- de la représentation interne des expressions,
- de l'implémentation du composant syntaxique,
- de l'implémentation du composant sémantique,
- du traitement des erreurs par PCL.

Ces choix sont détaillés complètement dans l'annexe 10 ("Choix d'implémentation").

2. Représentation interne des expressions

Malgré les avantages d'une représentation interne basée sur la syntaxe abstraite, cette solution n'a pas pu être retenue.

En effet, il n'a pas été possible de remettre en question le choix préexistant dans le projet DAIDA, qui consiste à stocker dans les Bases de Connaissances des expressions PCL sous leur forme concrète.

On a donc dû choisir une représentation interne basée sur la syntaxe concrète de PCL.

De plus, pour des raisons internes au projet, il n'a pas été possible non plus d'isoler cette représentation à l'intérieur d'un type abstrait, ce qui aurait protégé le programme des modifications éventuelles de la syntaxe concrète.

Les composants syntaxique et sémantique manipulent donc les expressions PCL directement sous leur forme concrète. On verra dans les points suivants les conséquences de ce choix.

3. Le composant syntaxique

On a dit au chapitre précédent que la technique d'implémentation de ce composant faisait elle aussi l'objet d'un choix préexistant dans le projet DAIDA, qui consistait à utiliser le PROLOG reader. Pour cette raison, la grammaire BNF du langage avait déjà été pensée en termes d'opérateurs.

L'implémentation du composant syntaxique a nécessité trois étapes, détaillées aux points suivants:

- Conversion de la grammaire BNF de PCL en une "grammaire à opérateurs hiérarchisée", de la manière décrite au chapire 3 de la troisième partie.
- Définition des opérateurs PROLOG correspondants.
- Implémentation d'un composant chargé de la vérification des conditions syntaxiques supplémentaires.

3.1. Conversion de la grammaire BNF

Cette conversion a pu être opérée facilement, vu la façon dont la grammaire BNF avait été définie.

Dans la grammaire résultante, la classification initiale des expressions (expressions primitives, expressions composées, ...) a fait place à une classification en niveaux, comme on peut le voir ci-dessous.

Grammaire convertie :

<expression> ::= <expr10>

<expr10> ::= <expr9> **OR** <expr10> | <expr9>

<expr9> ::= <expr8> **AND** <expr9> | <expr8>

<expr8> ::= **NOT** <expr8> | <expr7>

<expr7> ::= <expr7> **PLUS** <expr6> | <expr7> **MINUS** <expr6> | <expr6>

<expr6> ::= <expr6> **TIMES** <expr5> | <expr6> **DIV** <expr5> | <expr5>

<expr5> ::= **SETOF** <expr5> | **AVG** <expr5> | **MIN** <expr5> |
MAX <expr5> | **SUM** <expr5> | **COUNT** <expr5> |
<expression ternaire> | <expr4>

<expr4> ::= <expr3> **WHERE** <expr4> | <expr3>

<expr3> ::= <expr2> **EQ** <expr2> | <expr2> **NE** <expr2> |
<expr2> **GT** <expr2> | <expr2> **GE** <expr2> |
<expr2> **ST** <expr2> | <expr2> **SE** <expr2> |
<expr2> **ISIN** <expr2> | <expr2> **SETEQ** <expr2> |
<expr2> **MEMBER** <expr2> |
<expr2> **INCLUDED** <expr2> | <expr2>

<expr2> ::= <expr1> **#** <expr2> | <expr1>

<expr1> ::= **?** <variable> | <expr0> | <ensemble> | <tuple>

<expr0> ::= <constante> | <variable> | <classe> | <slot> | **(** <expression> **)**

<constante> ::= *constante PROLOG*

<variable> ::= *atome PROLOG*

<classe> ::= *atome PROLOG*

<slot> ::= *atome PROLOG*

<ensemble> ::= **[** <constante>$^{1..\infty}$ **]**

<tuple> ::= **(** <expression>$^{1..\infty}$ **)**

<expression ternaire> ::= <mot 1> <expr2> <mot 2> <expr2> <mot 3> <expr4>

<mot 1> ::= **EXIST** | **FORALL**

<mot2> ::= **MEMBER** | **INCLUDED**

<mot3> ::= **WITH**

3.2. Définition des opérateurs PROLOG

Cette définition a été établie systématiquement selon les règles présentées au chapitre 3 de la troisième partie, à partir de la grammaire présentée ci-dessus, et à partir de la définition des opérateurs concrets reprise en annexe 8 ("Choix d'une syntaxe concrète").

3.3. Vérification des conditions supplémentaires

Ce composant consiste en une procédure PROLOG, qui vérifie les conditions syntaxiques décrites par la grammaire attribuée définissant la correction syntaxique des expressions (annexe 5), notamment par l'intermédiaire du langage PROBE.

Elle reçoit en entrée un terme PROLOG, construit automatiquement par le PROLOG reader à partir d'une chaîne de caractères. Si ce terme représente bien une expression PCL correcte, la procédure réussit. Sinon, elle échoue en positionnant un code d'erreur (voir le point 5 de ce chapitre).

En outre, cette procédure a été dotée d'une fonctionalité supplémentaire, qui consiste à fournir la "liste des relations de dépendances"[1] des expressions qu'elle analyse. Cette liste peut en effet être facilement établie lors de l'analyse syntaxique. Cette notion est développée dans l'annexe 6: "Relations de dépendances".

On remarque qu'étant donné le choix de manipuler les expressions PCL sous leur forme concrète, le composant syntaxique que l'on vient de décrire n'effectue aucune traduction des expressions analysées en représentations internes, mais se contente de vérifier leur correction syntaxique.

---

[1] La "liste des relations de dépendance" d'une expression est définie comme la liste des noms des classes citées dans l'expression, ainsi que des couples (slot , nom de classe le contenant) pour tout slot qui y apparaît.

Les informations importantes issues de cette analyse, telles que la détermination du type des sous-expressions, ne sont donc pas mémorisées à l'attention du composant sémantique.

4. Le composant sémantique

Ce composant consiste en une procédure PROLOG, qui évalue des expressions PCL syntaxiquement correctes en fonction des règles établies par la grammaire attribuée définissant la sémantique (annexe 7).

Afin d'effectuer cette évaluation, la procédure doit refaire elle-même une analyse syntaxique presque complète, destinée à retrouver la structure de l'expression, ainsi que le type de ses sous-expressions. En fonction de ces informations, elle tente ensuite de déterminer la valeur de l'expression. Si elle y réussit, elle renvoie cette valeur. Sinon, elle positionne un code d'erreur (voir le point 5 de ce chapitre).

Cette procédure possède donc de grandes similitudes avec la procédure qui implémente le composant syntaxique, à tel point qu'il est tentant de l'utiliser directement pour évaluer une expression quelconque, sans effectuer d'analyse syntaxique explicite préalable...

5. Le traitement des erreurs

Le contrôle d'erreur est géré de manière similaire par tous les composants du programme.

Une erreur intervenant lors du traitement d'une expression PCL a pour effet la mise-à-jour d'une variable globale appelée PCLrc (pour "**PCL r**eturn **c**ode"), que les programmes appelants pourront consulter.

Cette variable globale peut prendre un certain nombre de valeurs significatives permettant de déterminer exactement la cause de l'erreur. Notamment, il est possible de distinguer les cas où l'erreur a été détectée par PCL, de ceux où elle a été détectée par PROBE, dont PCL utilise les services.

La liste des codes d'erreurs et de leur signification est reprise dans l'annexe 10 ("Choix d'implémentation").

Conclusion

Ce mémoire avait donc pour objectif la définition et l'implémentation du langage PCL, qui constitue l'un des composants de l'environnement de prototypage développé par le BIM dans le cadre du projet DAIDA.

Pour réaliser ce travail, on a établi un cadre théorique précis et détaillé, pouvant servir de base à la définition systématique du langage, ainsi qu'à son implémentation.

Ceci a permis de définir complètement le langage PCL, ainsi que le méta-modèle conceptuel des Bases de Connaissances qu'il permet de consulter. Une implémentation de ce langage a également été réalisée.

Il est possible d'envisager un certain nombre d'extensions de ce travail, tant du point de vue de la définition de PCL, que de celui de son implémentation.

D'abord, il serait possible d'étendre les capacités d'expression de PCL, en élargissant la notion d'ensemble explicite de valeurs, et en autorisant l'introduction d'appels à des procédures PROLOG définies par l'utilisateur. Ces extensions sont décrites en détail à l'annexe 9 ("Restrictions et extensions de la syntaxe abstraite"). Elles ont un impact important sur la définition du langage, car elles nécessitent une révision profonde de sa syntaxe abstraite.

Ensuite, l'implémentation du langage pourrait également être améliorée. D'une part, le programme pourrait être doté d'une architecture plus robuste, évitant la dissémination des informations relatives à la syntaxe concrète dans ses différents composants (voir le chapitre 3 de la troisième partie). D'autre part, l'utilisation d'une technique d'analyse syntaxique plus puissante que celle du PROLOG reader permettrait un meilleur contrôle des erreurs lors de la phase de tokenisation. Ceci améliorerait la qualité du programme actuel.

En conclusion, les résultats concrets obtenus par ce mémoire ont montré l'utilité de disposer d'un cadre théorique solide pour définir et implémenter un langage formalisé.

Ils ont également démontré l'applicabilité de tels outils théoriques à la construction effective d'un système informatique dans un "environnement de production".

Références bibliographiques

Références bibliographiques

[A.K.-NASR]
Hassan AÏT-KACI, Roger NASR
"Logic and Inheritance"
Communications of the ACM, 1986

[BIM]
J.L. Binot, W. Burgard, I. de Zegher, D. Donner, G. Michaux
"Integration of a frame based extension into a Prolog environment"
Protos, subproject A, Research paper, BIM 88

[BOBR]
Daniel G. Bobrow, Mark J. Stefik
"Loops: an object-oriented programming system for Interlisp"
Xerox PARC, 1982

[BOR-MYL-WONG]
Alexander Borgida, John Mylopoulos, Harry K.T. Wong
"On the conceptual modeling : Generalization/Specialization as Basis for Software Specification"
Springer-Verlag 1984

[CCI-BIM-UF]
CCI, BIM, UF
"Final version on TDL design, Deliverable DES1.2"
ESPRIT Project 892 DAIDA
Research paper, September 1987

[CLOCK-MELL]
W.F. Cloksin, C.S. Mellish
"Programming in Prolog", Second Edition.
Springer-Verlag
Berlin Heidelberg New-York Tokyo 1984

[COE-COT]
Helder Coelho, José C. Cotta
"Prolog by Example", How to Learn, Teach and Use It.
Springer-Verlag 1988

[COX]
Brad J. Cox
"Object-Oriented programming: an evolutionnary approach"
Addison-Wesley, Reading (Mass.), 1986

[DEV]
Y. Deville
"A Methodology for Logic Program Construction"
Ph-D thesis, Namur, 1987

[DR-N-R]
P. De Raedemacker, V.Nachtergaele, J. Rulkin
"La programmation logique"
Travail réalisé dans le cadre du cours de C. Cherton, 1988

[FIK-KEH]
Richard Fikes, Tom Kehler
Special Section : "The role of frame-based representation in reasoning"
Communications of the ACM, Sept. 85, vol 28, no 9

[GALL]
Hervé Gallaire
"La représentation des Connaissances"
La Recherche, no 170, octobre 85.

[GREEN]
Sol J. Greenspan
"Requirements Modeling : A Knowledge Representation Approach to Software Requirements Definition"
Thesis, Dep. of Computer Science, University of Toronto, 1984

[HOFST]
Douglas Hofstadter
"Gödel Escher Bach, Les Brins d'une Guirlande Eternelle"
Version française de Jacqueline Henry et Robert French
InterEditions, 1985

[HUB-VAR]
Martin Huber, Igor Varsek
"Extended Prolog for Order-Sorted Resolution"
Symposium on Logic Programming, IEEE, 1987

[JARKE]
Matthias Jarke, DAIDA Team
"The DAIDA Environment for Knowledge-Based Information Systems Development"
Abstract, ESPRIT PROJECT 892 (DAIDA), 1988

[KNU]
Donald E. Knuth
"Semantics of Context-free Languages"
California Institute of Technology
Mathematical Systems Theory, vol 2, no 2
Springer-Verlag, New-York

[KORNF]
W. A. Kornfeld
"Equality for Prolog",
Proc. IJCAI, p. 514-519, 1983

[LECH]
B. Le Charlier
"Théories des programmes"
Notes de cours, 1987-1988, Namur

[LER]
H. Leroy
"Compilation, m.a."
Notes de cours, 1987-1988, Namur

[LIV]
C. Livercy,
Nom collectif de : J.P. Finance, M. Grandbastien, P. Lescanne, P. Marchand, R. Mohr, A. Quéré, J.L. Rémy
"Théories des programmes, schémas, preuves, sémantique"
préface C.Pair, Dunod

[MARC]
Claudie Marcus
"Prolog Programming"
Addison Wesley 1986
Chapter 7 : Techniques for Language Processing.

[MEIR-TR-VEN]
E. Meirlaen, J.M. Trinon, R. Venken
"An object-based prototyping workbench in prolog"
Research paper, BIM 1988

[MEIR-TR]
E. Meirlaen, J.M. Trinon
"Mapping TDL/SML to Prolog : PROBE and PCL"
ESPRIT Project 892 DAIDA
Research paper, BIM 22th april 88

[MEY]
Bertrand Meyer
"Object-Oriented Software Construction"
Prentice Hall, 1988

[MOON]
David A. Moon
"Object-Oriented programming with Flavors"
ACM Conference on Object-Oriented programming, Systems, Languages and Applications,
Portland (Oreg.), 1986

[MYL-LEV]
John Mylopoulos, Hector J. Levesque
"On the conceptual modeling : An overview of Knowledge Representation"
Springer-Verlag 1984

[SHAP]
Ehud Shapiro
"A Subset of Concurrent Prolog and Its Interpreter",
Internal Report,
Weizmann Institute of Science, Rehovot, Israel, 1983

[SMALL]
A. Goldberg, D. Robson, D. Ingalls
"Smalltalk 80 : The language and its implementation"
Chapter 1 : Objects and Messages
Addison-Wesley, 1983

[STER-SHAP]
Leon Sterling, Ehud Shapiro
"The Art of Prolog, Advanced Programming Techniques"
MIT Press Series in Logic Programming, 1986

[STROU]
Bjanre Stroustrup
"The C++ programming Language"
Addison-Wesley, Menlo Park (Calif.), 1986

[TENN]
R. D. Tennent
"Principles of Programming Languages"
C.A.R. Hoare Serie, Prentice Hall, 1981.

[ZAN]
Carlo Zaniolo
"Object-Oriented programming in Prolog"
Symposium on Logic Programming, IEEE, 1984

Définition d'un langage de consultation pour une Base de Connaissances "orientée-objet"

(Annexes)

Véronique Nachtergaele

Mémoire réalisé sous la
direction du Professeur B. Le Charlier
en vue de l'obtention du titre de
Licencié et Maître en Informatique.

Namur Juin 1989.

Table des matières

Annexe 1 : Méta-modèle conceptuel, définition 1

1. Introduction 2
2. Présentation intuitive 3
3. Syntaxe abstraite 5
   - 3.1. Concepts décrits 5
   - 3.2. Le niveau de coherence ZERO 5
   - 3.3. Le niveau de cohérence UN 8
   - 3.4. Le niveau de cohérence DEUX 9
   - 3.5. Le niveau de cohérence TROIS 11
   - 3.6. Le niveau de cohérence QUATRE 12
4. Sémantique 13
   - 4.1. Le niveau de cohérence ZERO 13
   - 4.2. Le niveau de cohérence UN 17
   - 4.3. Le niveau de cohérence DEUX 18
   - 4.4. Le niveau de cohérence TROIS 21
   - 4.5. Le niveau de cohérence QUATRE 25

Annexe 2 : Schéma des niveaux de cohérence 26

1. Schéma des niveaux de cohérence 27
2. Description par niveau de cohérence 27
   - 2.1. Le niveau de cohérence ZERO 28
   - 2.2. Le niveau de cohérence UN 28
   - 2.3. Le niveau de cohérence DEUX 29
   - 2.4. Le niveau de cohérence TROIS 30
   - 2.5. Le niveau de cohérence QUATRE 31

Annexe 3 : Contexte de définition d'une expression 32

1. Presentation du contexte de définition des expressions 33

Annexe 4 : Description intuitive des expressions 35

1. Introduction 36
2. Description intuitive du langage 37
3. Les expressions de base 38
4. Les opérateurs 41
   - 4.1. Opérateur associant un slot à une classe 41
   - 4.2. Opérateur formant un ensemble 41

4.3. Opérateur posant une condition sur un objet 41
4.4. Opérateurs logiques 42
4.5. Opérateurs de comparaison 45
4.6. Opérateurs ensemblistes 46
4.7. Operateurs définissant des fonctions agregees 47
4.8. Opérateurs arithmétiques 49

Annexe 5 : Syntaxe abstraite 50

1. Introduction 51
2. Type d'une expression 51
   2.1. Expressions de base 51
   2.2. Compatibilité de type et type résultant des opérateurs 53
3. Description de la grammaire 63
   3.1. Attributs utilisés 63
   3.2. Sémantique des attributs 64
4. Grammaire attribuee 67
   4.1. Correction syntaxique d'une expression 67
   4.2. Correction syntaxique d'un slot 108
   4.3. Correction syntaxique d'une requête 109

Annexe 6 : Relations de dépendance 110

1. Notion de relations de dépendance 111
2. Définition des relations de dépendance 113
   2.1. Relations de dépendance d'une expression 113
   2.2. Relations de dépendance d'une définition de slot 119
   2.3. Relations de dépendance d'une requête 119

Annexe 7 : Sémantique 120

1. Introduction 121
2. Description de la grammaire 121
   2.1. Attributs utilisés 121
   2.2. Sémantique des attributs 122
3. Grammaire attribuée 125
   3.1. Sémantique d'une expression 125
   3.2. Sémantique d'un slot 170
   3.3. Sémantique d'une requête 170
4. Contrainte sur le modèle conceptuel 172

Annexe 8 : Choix d'une syntaxe concrète 173

1. Introduction 174
2. Conventions d'écriture 174
3. Présentation de la syntaxe 174
4. Syntaxe 176
   4.1. Expressions de base 176
   4.2. Opérateurs 177
   4.3. Mot défini 212

Annexe 9 : Restrictions et extensions 213

1. Introduction 214
2. Parallélisme syntaxe abstraite / concrète 214
   2.1. Parallélisme pour les expressions de base 214
   2.2. Parallélisme pour les opérateurs 215
3. Restriction de la syntaxe abstraite 218
4. Extensions possibles 219

Annexe 10 : Choix d'implémentation 221

1. Introduction 222
2. Définition "Prolog" des opérateurs 223
3. Implémentation de la syntaxe concrète 224
   3.1. Classification des procédures 224
   3.2. Schéma de spécification d'une procedure 225
   3.3. Remarques générales 226
   3.4. Présentation des procédures 227
   3.5. Codes d'erreurs 245

Annexe 11 : Listing 251

Méta-modèle conceptuel,
Définition

Annexe 1

1. Introduction

La premiere partie de ce document consiste en une description intuitive de tous les concepts décrits dans un modèle conceptuel.

La seconde partie est la définition de la Base de Connaissances, qui comprend deux parties :

- la syntaxe abstraite du méta-modèle conceptuel de la Base de Connaissances.

  Cette définition syntaxique du méta-modèle conceptuel est donnée suivant une structure hiérarchique de niveaux de cohérence.

- la sémantique, décrivant l'ensemble de toutes les instances possibles du méta-modèle conceptuel.

  Cette partie définit la sémantique associée aux concepts et relations qui décrivent le méta-modèle conceptuel.

  Cette description sémantique de la Base de Connaissances est également donnée suivant une structure de niveaux de cohérence correspondant aux niveaux de cohérence de la syntaxe du méta-modèle conceptuel.

Le méta-modèle conceptuel est donc décrit selon une structure hiérarchique. Chaque niveau est numéroté niveau **ZERO**, **UN**, **DEUX**, ... et le niveau **i** est un raffinement (une restriction) du niveau **i-1**.

2. Présentation intuitive

Cette première partie décrit intuitivement les concepts généralement utilisés dans la description de tout modèle conceptuel de Base de Connaissances.

Ces concepts sont les suivants :

- classe
- instance d'une classe
- slot
- facette d'un slot
- metaclasse
- relation IS-A (notion de superclasse)

Une Base de Connaissances est composée d'un modèle conceptuel et d'une collection d'objets appelée instance du modèle.

Un modèle conceptuel est une description d'un ensemble de classes et de relations.

Une classe dénote une collection d'éléments. Chaque classe possède un nom ainsi qu'un ensemble de descriptions de propriétés qui doivent être satisfaites par chaque élément de la classe.

Un élément d'une classe est appelé instance de la classe.

Chaque élément de la description d'une classe est appelé slot.

Les slots sont répartis en deux catégories :

-"slots de propriété" :

ils peuvent posséder une valeur qui désigne une (un ensemble d') instance(s) ; ils expriment une relation sémantique entre classes.

- "slot de contrainte" :

ils expriment une contrainte sur les valeurs possibles des slots de propriété d'une classe ; ils restreignent ainsi l'ensemble des valeurs possibles de ces slots de propriété.

Chaque slot est décrit par un ensemble de facettes : ces facettes permettent de préciser les propriétés de ce slot.

Chaque classe appartient a une seule metaclasse, les principales métaclasses étant "primitive", "enumerated", "range", "entity", "aggregate".

Une relation IS-A peut être décrite entre deux classes "A" et "B". Une telle relation signifie que toutes les instances de "A" sont également instances de "B".

Un exemple peut aider a clarifier ces notions !

Soit une classe "Personne". La description de la classe est la collection suivante de slots : "Nom", "Prénom", "Age", chaque slot étant caractérisé par une liste de facettes.

La classe "Personne" appartient à la métaclasse "entity".

Une instance de cette classe est décrite, par exemple, par les trois valeurs {Smith, Jo, 38}.

Soit une deuxième classe "Employé", telle que sa description soit la collection de slots "Nom", "Prénom", "Age", "Travaille sur projet".

On peut décrire une relation IS-A entre les classes "Employé" et "Personne", exprimant ainsi que toutes les instances de la classe "Employé" sont également des instances de la classe "Personne".

3. Syntaxe abstraite

La description syntaxique du méta-modèle conceptuel est présentée graduellement, définissant ainsi cinq niveaux de cohérence syntaxique, chacun d'eux raffinant le précédent.

3.1. Concepts décrits

Voici la liste des concepts décrits dans les sections suivantes :

- classe
- slot
- facette d'un slot
- métaclasse
- relation IS-A entre classes (notion de superclasse)

3.2. Le niveau de cohérence ZERO

Ce niveau comprend les concepts les plus simples du modèle.

3.2.1. Définition de facette

Une définition de facette est un couple (*facetname* , *value*), où le nom de facette *facetname* doit être l'un des suivants:

- def
- default
- categ
- card
- comment
- presence
- reverse

Convention :

On désignera le couple (*facetname* , *value*) par la formulation "facette *FACETNAME* .

Ce type de convention sera également utilisé pour les autres concepts.

3.2.2. Définition de slot

Une définition de slot est :

- un nom de slot
- un ensemble de définitions de facette,

  tel que chaque nom de facette est unique dans l'ensemble de définitions de facette.

3.2.3. Définition de classe

Une définition de classe est :

- un nom de classe
- un ensemble de définitions de slot,

  tel que chaque nom de slot est unique dans l'ensemble des définitions de slot.

3.2.4. Définition de relation IS-A

Une définition de relation IS-A est :

un couple de classes (Cl_1, Cl_2).

Une telle définition est notée :

- **Cl_1 --"IS-A"--> Cl_2**

et se lit :

- Cl_1 est une sous-classe de Cl_2
- Cl_2 est une superclasse de Cl_1

Comme conséquence de la relation IS-A, on peut considérer deux types de slots :

- slot propre S_j d'une classe Cl_i :

  définition de slot explicitement **définie dans** la définition de classe Cl_i.

- slot hérité S_j d'une classe Cl_i :

définition de slot **définie pour** la définition de classe Cl_k, et devenu slot de la définition de classe Cl_i comme conséquence de la définition de relation IS-A (Cl_i, Cl_k).

Note :

Si un slot propre **défini dans** la classe Cl_i a le même nom qu'un slot hérité de la classe Cl_i (**défini pour** une autre classe Cl_k), le slot propre est le seul des deux qui est pris en compte et qui appartient à la définition de classe. Le slot hérité disparaît.

Si un slot hérité de la classe Cl_i (**défini pour** la classe Cl_k) a le même nom qu'un autre slot hérité de la classe Cl_i (**défini pour** une autre classe Cl_j), les deux slots sont sans effet et un slot propre doit être **défini dans** la classe Cl_i, ce slot propre ayant le même nom que les deux autres.

3.2.5. Nom de métaclasse

Un nom de metaclasse est l'un des suivants :

- primitive
- enumerated
- range
- entity
- aggregate

3.2.6. Le modèle conceptuel

Le modèle conceptuel est défini par :

- un ensemble de définitions de classe,
  tel que chaque nom de classe est unique dans l'ensemble.
- un ensemble de définitions de relations IS-A,
- un ensemble de couples (*ClassName* , *METAclassName*),
  tel que chaque nom de classe *ClassName* de l'ensemble des définitions de classe apparaît une et une seule fois comme premier élément du couple,
  et *METAclassName* est un nom de métaclasse.

Note :

On désignera l'ensemble des classes primitives, enumerated, range sous le nom générique de classes dites "basic".

De manière analogue, on désignera l'ensemble des classes entity, aggregate sous le nom générique de classes dites "non-basic".

3.3. Le niveau de cohérence UN

Un modèle conceptuel de niveau de coherence UN est un modèle conceptuel de niveau de cohérence ZERO, respectant les conditions suivantes :

Une définition de facette categ est de la forme (categ, *value*), où *value* est une des valeurs suivantes :

- changing
- unchanging
- derivation
- initcond
- finalcond
- invariant

Note :

Le nom générique de "slot de propriété" regroupe toutes les définitions de slot qui contiennent une facette (categ, changing), (categ, unchanging) ou (categ, derivation).

Le nom générique de "slot de contrainte" regroupe, quant à lui, toutes les définitions de slot qui contiennent une facette (categ, initcond), (categ, finalcond) ou (categ, invariant).

Une définition de facette card est de la forme (card, m-n), où m est un entier,

n est un entier non nul ou la valeur U,

m <= n ou n = U

Une définition de facette comment est de la forme (comment, *value*), où *value* est un texte contenant au maximum 256 caractères.

Une définition de facette presence est de la forme (presence, *value*), où *value* est l'une des valeurs suivantes :

- mandatory
- optional

Une définition de facette reverse est de la forme (reverse, *slotname*), où *slotname* est un nom de slot d'une définition de slot appartenant à une définition de classe du modèle conceptuel.

3.4. Le niveau de cohérence DEUX

Un modèle conceptuel de niveau de cohérence DEUX est un modèle conceptuel de niveau de cohérence UN, respectant les conditions suivantes :

L'ensemble des définitions de relations IS-A définit un graphe sans cycle construit de la manière suivante :

- chaque noeud est un nom de classe.
- il existe un arc entre deux noeuds A et B (de A vers B), si il existe un couple (A, B) dans l'ensemble des définitions de relations IS-A

Les classes enumerated contiennent une seule définition de slot, qui est un slot extension ayant les facettes suivantes :

- une facette (def, *value*), où *value* est une liste d'atomes,
- éventuellement une facette comment.

Les classes range contiennent seulement deux définitions de slot, (les deux seuls slots possibles) :

le premier étant défini par :

- le nom extension
- la facette (def, *value*) où *value* est de la forme :

  m-n, ou m,n entiers, et m < n
- eventuellement une facette comment.

et le second étant défini par :

- le nom type
- la facette (def, *value*) où *value* est l'une des valeurs :

  integer
  real
- éventuellement une facette comment.

Les classes primitive sont définies par un ensemble vide de définitions de slot.

La facette def est la seule facette obligatoire pour chaque slot d'une classe enumerated, range, ou non-basic.

La facette card ne possède de signification que pour les définitions de slot contenant également une facette (categ, changing) ou (categ, unchanging), et appartenant à une classe dite non-basic.

La facette presence est uniquement prise en compte pour les définitions de slot contenant également une facette (categ, changing), (categ, unchanging) ou (categ, derivation), et appartenant à une classe dite non-basic.

La facette reverse est uniquement prise en compte pour les définitions de slot contenant également une facette (categ, changing) ou (categ, unchanging), et appartenant à une classe dite non-basic.

La facette default est uniquement prise en compte pour les définitions de slot contenant également une facette (categ, changing), (categ, unchanging) ou (categ, derivation), et appartenant à une classe dite non-basic.

3.5. Le niveau de cohérence TROIS

Un modèle conceptuel de niveau de cohérence TROIS est un modèle conceptuel de niveau de cohérence DEUX, respectant les conditions suivantes :

Ce niveau de cohérence TROIS consiste principalement en la définition syntaxique d'une expression PCL.

Toutes les définitions de slot d'une classe dite non-basic doivent contenir une définition de facette (def, *value*),

> tel que, *value* est une expression PCL syntaxiquement correcte, respectant les règles de cohérence (des expressions) du méta-modèle conceptuel de la Base de Connaissances.
>
> Une expression PCL syntaxiquement correcte est une expression PCL qui satisfait la syntaxe des expressions PCL au niveau de cohérence TROIS.

Pour toutes les définitions de slot d'une classe dite non-basic, qui contiennent une facette (default, *value*) :

> *value* doit être du type défini par l'expression PCL définissant la facette def du slot.

Pour toutes les définitions de slot d'une classe dite non-basic, qui contiennent une facette (reverse, *value*) :

> Soit :
>
> - la classe Cl_1 contient un slot S_1.
> - le slot S_1 contient une facette (def, ValueS1),
>
> telle que ValueS1 est une expression PCL référençant la classe Cl_2.
> - la classe Cl_2 contient un slot S_2.
> - le slot S_2 contient une facette (def, ValueS2),
>
> telle que ValueS2 est une expression PCL référençant la classe Cl_1 ou une superclasse de Cl_1.
>
> ALORS, le slot S_1 slot peut contenir une facette (reverse, S_2).

Un exemple de règles de cohérence des expressions est :

> "Tous les slots d'une expression PCL concernant une Base de Connaissances particulière doivent être des slots décrits dans le modèle conceptuel de la Base de Connaissances."

3.6. Le niveau de cohérence QUATRE

Un modèle conceptuel de niveau de coherence QUATRE est un modèle conceptuel de niveau de cohérence TROIS , respectant les conditions suivantes :

Le niveau de cohérence TROIS consistant principalement en la définition syntaxique d'une expression PCL, ce niveau QUATRE restreint la définition des expressions PCL dans les cas particuliers des facettes def des définitions de slot (de propriété ou. de contrainte), ainsi que les requêtes.

3.6.1. Slots de propriété

L'expression PCL exprimée par la facette def d'un slot de propriété est définie par une expression PCL, syntaxiquement correcte, et dont la forme est la suivante :

Pour les slots fonctionnels

- *classname*
- **COND(***classname, expr* **)**

Pour les slots relationnels

- **SET(** *classname* **)**
- **SET(COND(** *classname, expr* **))**

3.6.2. Slots de contrainte

L'expression PCL exprimée par la facette def d'un slot de contrainte est définie par une expression PCL quelconque, syntaxiquement correcte, et dont le type est boolean.

3.6.3. Requêtes

Toute expression PCL, syntaxiquement correcte, exprimant une requête est définie par une expression de la forme suivante :

(t_1, ... , t_n)

où pour tout i : t_i est de la forme :

- **ISofClass(** *slotname, expr* **)**
- **SET(** *expr* **)**
- **COND(** *expr1, expr2* **)**
- **FUNCT(** *expr* **)**
- **AOp(** *expr1, expr2* **)**

4. Sémantique

Cette partie concerne la description sémantique de la Base de Connaissances.

La présentation de la sémantique suit le schéma des cinq niveaux utilisé pour la définition syntaxique du méta-modèle conceptuel.

4.1. Le niveau de cohérence ZERO

La sémantique de la Base de Connaissances est exprimée en définissant la sémantique de chaque objet du méta-modèle conceptuel.

Le niveau de cohérence ZERO définit la sémantique des concepts suivants :

- métaclasse
- définition de classe
- définition de relation IS-A
- définition de slot

4.1.1. Sémantique associée au concept de métaclasse

Un couple (*ClassName* , *MetaClassName*) est l'expression de l'appartenance de la classe de nom *ClassName* à la métaclasse *MetaClassName* .

La notion de métaclasse correspond à la description d'un ensemble générique de définitions de classe décrites dans le modèle conceptuel. Chaque classe d'un tel ensemble générique de classes possède les même fonctionnalités que toutes les autres classes de l'ensemble, et les même traitements peuvent être appliqués à chaque classe de cet ensemble.

La métaclasse primitive contient les seules classes predéfinies suivantes :

- Integer
- Integer(m,n)
- String
- String(m,n)
- Real
- Boolean

La métaclasse range regroupe les définitions de classe qui sont des ensembles restreints de nombres (entiers ou réels).

La métaclasse enumerated regroupe les classes qui sont des ensembles de valeurs (chaque élément représente une seule valeur).

Les métaclasses entity and aggregate regroupent toutes deux des classes définies par un ensemble de définitions de slot quelconques. Il n'existe aucune différence au niveau du langage PCL entre ces deux métaclasses ; la distinction de l'appartenance d'une classe à l'une ou l'autre métaclasse aurait pu être introduite dans un méta-modèle englobant ce méta-modèle conceptuel, cette distinction ne concernant qu'une différence d'implémentation. [1]

4.1.2. Sémantique associée à une définition de classe

A chaque classe Cl_i, on associe un ensemble $E[Cl_i]$, qui est l'ensemble des instances possibles de la définition de classe Cl_i. Il en résulte donc que chaque élément de l'ensemble $E[Cl_i]$ doit satisfaire la définition de classe Cl_i.

On remarquera que cet ensemble $E[Cl_i]$ est plus ou moins défini selon que la classe Cl_i appartient à l'une ou l'autre métaclasse.

Chacun des éléments de l'ensemble $E[Cl_i]$ est appelé "instance de la classe" Cl_i. De plus, chaque instance possède un slot implicite de nom **identifier** qui permet de la distinguer des autres instances (de la classe, ainsi que des autres classes).

Pour les définitions de classes Cli entity ou aggregate :

(classes dites non basic)

Considérons la définition de classe Cl_i pour laquelle les slots $S_1,\ldots,S_n$ sont définis.

pour tout k, $F[S_k]$ = { valeurs possibles du slot S_k }.

$E[Cl_i]$ = { valeurs du slot implicite **identifier** pour les instances de la classe Cl_i, où une instance est une référence à un élément du produit cartésien des $F[S_k]$ pour tout k}.

---

[1] La distinction entre les métaclasses entity et aggregate a été introduite pour permettre la description d'attributs décomposables dans les niveaux supérieurs (au niveau du langage TDL) du projet ESPRIT "DAIDA, ESPRIT Project 892", duquel ce modèle est inspiré. En fait, dans ce travail, aucune distinction n'est faite entre les deux types de métaclasses.

4.1.3. Sémantique d'une relation IS-A

La sémantique associée à une relation IS-A, notée

$$\mathbf{Cl_i \text{ ---"IS-A"---> } Cl_k}$$

fournit la contrainte suivante :

$$\mathbf{E[Cl_i] \subseteq E[Cl_k]}$$

Note :

Comme conséquence de cette contrainte d'inclusion, on peut définir les notions suivantes :

- instance propre d'une classe :

  une instance *ins* d'une classe Cl_i est dite instance propre de la classe Cl_i **si**

  1. $ins \in E[Cl_i]$
  2. $\forall$ k, ins $\notin E[Cl_k]$, sauf si $E[Cl_i] \subset E[Cl_k]$

- instance spécialisée d'une classe :

  une instance *ins* d'une classe Cl_i est dite instance spécialisée de la classe Cl_k **si**

  1. *ins* est une instance (propre ou spécialisée) de Cl_i
  2. $E[Cl_i] \subset E[Cl_k]$

4.1.4. Sémantique associée à un slot

La sémantique d'une définition de slot est exprimée en associant au slot une fonction, définie comme suit :

Soit une classe Cl_i contenant la définition de slot S_j :

- l'ensemble de départ est l'ensemble $E[Cl_i]$
- l'ensemble d'arrivée est l'ensemble $F[S_j]$

  $F[S_j]$ = { valeurs possibles du slot S_j }

On utilisera la notation suivante :

$S_j( )$: $E[Cl_i]$ ----------> $F[S_j]$

ins ~~~~~~~~> S_j(ins)

où

S_j est le nom d'un slot de la classe Cl_i

S_j( ) est la fonction associée au slot S_j

E[Cl_i] est l'ensemble des instances de la classe Cl_i

F[S_j] est l'ensemble de toutes les valeurs possibles pour le slot S_j , qui est défini dans la classe Cl_i

4.1.5. Sémantique du modèle conceptuel

La sémantique du modèle conceptuel est donc définie en associant à chaque classe du modèle l'ensemble des instances de cette classe, et en associant à chaque slot une fonction définissant la valeur de ce slot pour toute instance d'une classe pour laquelle ce slot est défini.

4.2. Le niveau de cohérence UN

La définition sémantique de la Base de Connaissances au niveau UN avec le méta-modèle conceptuel est la définition sémantique de la Base de Connaissances au niveau ZERO avec le méta-modèle conceptuel, respectant les règles supplémentaires ci-dessous.

Ces regles concernent la semantique des definitions de facettes.

Il est important de voir que chaque définition de facette permet de préciser des informations concernant la définition de slot dans laquelle elle intervient, et concernant la valeur que le slot peut prendre.

4.2.1. Sémantique associée à la facette (comment, value)

Aucune sémantique particulière n'est associée à cette facette ; elle est laissée a la discrétion du concepteur.

4.2.2. Sémantique associée à la facette (presence, value)

Selon la valeur du second argument du couple (presence,value) :

- la valeur mandatory précise que le slot doit posséder une valeur pour chaque instance. La fonction $S_j(\ )$ définissant la sémantique du slot est une fonction totale, partout définie.

  $S_j(\ )$: $E[Cl_i]$ ----------> $F[S_j]$

  ins ~~~~~~~~> S_j(ins)

- la valeur optional précise que le slot peut ne posséder aucune valeur pour certaines instances. La fonction $S_j(\ )$ définissant la sémantique du slot est une fonction partielle

  $S_j(\ )$: $E[Cl_i]$ ----------> $F[S_j]$

  ins ~~~~~~~~> S_j(ins)

  On peut également définir une fonction $S_j^*(\ )$ totale, partout définie, ayant même sémantique que $S_j(\ )$, et associant la valeur undef(ined) à toute instance pour laquelle le slot ne possède pas de valeur définie :

  $S_j^*(\ )$: $E[Cl_i]$ ----------> $F[S_j]$ + { undef(ined) }

  ins ~~~~~~~~> S_j(ins)

4.3. Le niveau de cohérence DEUX

La définition semantique de la Base de Connaissances au niveau DEUX avec le méta-modèle conceptuel est la définition sémantique de la Base de Connaissances au niveau UN avec le méta-modèle conceptuel, respectant les règles supplémentaires ci-dessous.

4.3.1. Sémantique associée aux classes dites basic

A. Pour les définitions de classe primitives Cli :

- Pour la classe *Integer* ,

  $E[Cl_i]$ = { x tel que x est entier}.

- Pour la classe *Integer(m,n)* ,

  $E[Cl_i]$ = {x tel que (10^{m-1} <= x) et (x < 10^{n} si n est entier)}.

- Pour la classe *String* ,

  $E[Cl_i]$ = { s tel que s est un string }.

- Pour la classe *String(m,n)* ,

  $E[Cl_i]$ = { s tel que (s est un string) et (s contient au moins m caractères) et (s contient au plus n caractères) }.

- Pour la classe *Real* ,

  $E[Cl_i]$ = { x tel que x est reel }.

- Pour la classe *Boolean* ,

  $E[Cl_i]$ = { TRUE, FALSE }.

B. Pour les définitions de classe enumerated Cl_i :

$E[Cl_i]$ = { atomes de la liste définissant la facette def du slot extension de la classe Cl_i }.

C. Pour les définitions de classe range Cl_i :

- Si le slot type possède une facette (def, integer), et que le slot extension possède une facette (def, m-n):

  où (m entier) et (n entier > m)

  ALORS,

  $E[Cl_i]$ = { x tel que (x entier), (m <= x) et (x <= n si n entier)}.

- Si le slot type possède une facette (def, real), et que le slot extension possède une facette (def, m-n) :

  où (m réel) et (n réel > m)

  ALORS,

  $E[Cl_i]$ = { x tel que (x réel) et (m <= x) et (x <= n si n réel)}.

4.3.2. Sémantique associée aux slots des classes dites non-basic

La sémantique d'une définition de slot a déjà été définie en associant au slot S_j une fonction, définie comme suit :

$S_j( )$: $E[Cl_i]$ ----------> $F[S_j]$

ins ~~~~~~~~> S_j(ins)

où

S_j est le nom d'un slot de la classe Cl_i

$S_j( )$ est la fonction associée au slot S_j

$E[Cl_i]$ est l'ensemble des instances de la classe Cl_i

$F[S_j]$ est l'ensemble de toutes les valeurs possibles pour le slot S_j , qui est défini dans la classe Cl_i

La définition de cette fonction $S_j( )$ peut être raffinée, en fonction des caractéristiques du slot S_j. Ce raffinement se traduit par une redéfinition de la valeur de l'ensemble F[Sj].

1. Sémantique des slots de propriété

Les slots de propriété peuvent être classés en deux catégories :

- les slots fonctionnels,

  slots dont la valeur est une valeur simple (un seul élément).

- les slots relationnels,

  slots dont la valeur est un ensemble de valeurs simples (un ensemble d'éléments).

A. Les slots fonctionnels

L'ensemble **$F[S_j]$** est un ensemble de valeurs simples.

B. Les slots relationnels

L'ensemble $\mathbf{F[S_j]}$ est de la forme suivante :

$$\mathbf{F[S_j]} = P(SV_j),$$

où SV_j est un ensemble de valeurs simples.

Note :

Il est important de remarquer qu'un slot relationnel peut posséder, pour une instance donnée, la valeur v = {}, aussi bien que valeur v = undef[ined].

La signification de chacune de ces deux valeurs possibles est radicalement différente.

La valeur v = {} est une valeur comme une autre, tandis que la valeur v = undef[ined] signifie que le slot ne possède pas de valeur pour cette instance.

2. Sémantique des slots de contrainte

L'ensemble $\mathbf{F[S_j]}$ est l'ensemble E[B] = { TRUE, FALSE }.

$S_j(\ )$: $E[Cl_i]$ ----------> E[B]

ins ~~~~~~~~> S_j(ins)

où

B désigne la classe Boolean primitive

Par analogie, un tel slot est appelé un slot fonctionnel.

4.4. Le niveau de cohérence TROIS

La définition semantique de la Base de Connaissances au niveau TROIS avec le méta-modèle conceptuel est la définition sémantique de la Base de Connaissances au niveau DEUX avec le méta-modèle conceptuel, respectant les règles supplémentaires ci-dessous.

A ce niveau, pour une Base de Connaissances particulière, on considère que tous les slots possèdent une valeur (fixée) pour chaque instance de chaque classe.

4.4.1. Sémantique associée à la facette (card, value)

La facette (card, *value*) caractérise le nombre de valeurs qu'un slot peut posséder pour chaque instance de la classe Cl_i contenant une facette (def, PCLexpression), tel que le type de l'expression PCL PCLexpression désigne un ensemble de valeurs, et pas seulement un seul élément.

L'ensemble $F[S_j]$ est alors défini comme suit :

$F[S_j]$ = { t-uples de valeurs possibles tel que $m <= t <= n$}.

4.4.2. Sémantique associée aux définitions de slot des classes dites non-basic

Ayant défini la sémantique d'une expression PCL, on peut raffiner la sémantique associée aux définitions de slot des classes dites non-basic, qui sont également des slots de propriété.

Ce raffinement est basé sur la connaissance de l'expression PCL donnée par le second argument de la facette (def, PCLexpression), et consiste en une redéfinition de l'ensemble $F[S_j]$.

Soit une définition de slot S_j d'une classe Cl_i. On peut donc considérer les deux catégories de slots suivantes :

1. **Les slots fonctionnels :**

L'ensemble $\mathbf{F[S_j]}$ est défini comme étant l'ensemble des instances de la classe référenciée par l'expression PCL de la facette (def, PCLexpression), de telle sorte que le type de la valeur $S_j(ins)$ soit le type de l'expression PCL.

Les slots de propriété dits "fonctionnels" sont des slots pour lesquels le type de l'expression PCL donnée par la facette def est l'un des types suivants :

- *classname*,

- integer,
- real,
- string.

2. Les slots relationnels :

L'ensemble $\mathbf{F[S_j]}$ est défini comme ayant l'une des formes suivantes :

1. $\mathbf{F[S_j]} = \{ X \in P(SV_j)$ tel que $m <= \#(X) <= n\}$

   si le slot Sj contient une facette (card, m-n),

   où $n > 1$ et $n \neq U$

2. $\mathbf{F[S_j]} = P(SV_j)$

   si le slot Sj ne contient pas de facette (card, m-n),

   ou contient la facette (card, 0-U).

L'ensemble $\mathbf{SV_j}$ est défini comme étant l'ensemble des instances de la classe référenciée par l'expression PCL de la facette (def, PCLexpression), de telle sorte que le type de la valeur S_j(ins) = { x tel que $x \in SVj$} soit le type de l'expression PCL.

Les slots de propriété dits "relationnels" sont des slots pour lesquels le type de l'expression PCL donnée par la facette def est l'un des types suivants :

- set(*classname*),
- set(integer),
- set(real),
- set(string).

4.4.3. Sémantique associée à la facette (reverse, value)

Soit :

- une classe Cl_1 pour laquelle est défini un slot S_1 .
- le slot S_1 contient une facette définition (def, ValueS1),

  telle que ValueS1 est une expression PCL référençant la classe Cl_2

$S_1( )$: $E[Cl_1]$ ----------> $F[S_1]$

ins ~~~~~~~~> $S_1(ins)$

- la classe Cl_2 pour laquelle est défini un slot S_2 .
- le slot S_2 contient une facette définition (def, ValueS2),

  telle que ValueS2 est une expression PCL référençant la classe Cl_1 ou une superclasse de cette classe Cl_1.

$S_2( )$: $E[Cl_2]$ ----------> $F[S_2]$

ins ~~~~~~~~> $S_2(ins)$

alors, le slot S_1 peut contenir une facette reverse (reverse, S_2) .

Selon la catégorie de slot :

Si S_1 et S_2 sont deux slots fonctionnels :

$\forall$ ins $\in$ $E[Cl_1]$: ins = $S_2(S_1(ins))$

$F[S_1] \subseteq E[Cl_2]$

Note :

$F[S_1]$ est restreint à l'ensemble des instances (propres ou spécialisées) de $E[Cl_2]$ pour lesquelles la définition de slot S_2 est la même que pour la classe Cl_2 .

Si S_1 est un slot fonctionnel et S_2 un slot relationnel :

$\forall$ ins $\in$ $E[Cl_1]$:

ins $\subseteq$ $S_2(S_1(ins))$

$F[S_1] \subseteq E[Cl_2]$

Note :

$F[S_1]$ est restreint à l'ensemble des instances (propres ou spécialisées) de $E[Cl_2]$ pour lesquelles la définition de slot S_2 est la même que pour la classe Cl_2 .

Si S_1 est un slot relationnel et S_2 un slot fonctionnel :

$\forall$ ins $\in$ $E[Cl_1]$:

$\forall$ x $\in$ S_1(ins), S_1(ins) $\neq$ undef : $S_2(x)$ = ins

$SV_1 \subseteq E[Cl_2]$

Note :

SV_1 est restreint à l'ensemble des instances (propres ou spécialisées) de $E[Cl_2]$ pour lesquelles la définition de slot S_2 est la même que pour la classe Cl_2 .

Si S_1 et S_2 sont deux slots relationnels :

$\forall$ ins $\in$ $E[Cl_1]$:

$\forall$ x $\in$ S_1(ins), S_1(ins) $\neq$ undef : ins $\subseteq$ $S_2(x)$

$SV_1 \subseteq E[Cl_2]$

Note :

SV_1 est restreint à l'ensemble des instances (propres ou spécialisées) de $E[Cl_2]$ pour lesquelles la définition de slot S_2 est la même que pour la classe Cl_2 .

4.4.4. Sémantique associée à la facette (default, value)

Le second argument de la facette default est considéré comme une valeur de slot (notée S_j(ins)) quand aucune autre valeur ne peut être évaluée pour le slot contenant une facette (categ, derivation), ou quand aucune autre valeur n'est enregistrée.

4.4.5. Sémantique des expressions PCL

Il est à noter que la sémantique des expressions PCL est définie sur une Base de Connaissances particulière (fixée, statique) qui doit être syntaxiquement et sémantiquement cohérente avec le niveau TROIS.

4.5. Le niveau de cohérence QUATRE

La définition semantique de la Base de Connaissances au niveau QUATRE avec le modèle conceptuel est la définition sémantique de la Base de Connaissances au niveau TROIS avec le modèle conceptuel, respectant les règles supplémentaires ci-dessous.

A ce niveau, le méta-modèle conceptuel ne définit, d'un point de vue syntaxique, que des restrictions concernant l'utilisation des epxressions PCL dans certains cas particuliers.

La sémantique de toute expression PCL étant définie au niveau TROIS, aucune définition sémantique ne doit donc être ajoutée à ce niveau.

Schéma
des niveaux de cohérence
du méta-modèle conceptuel

Annexe 2

1. Schéma des niveaux de cohérence

	Cohérence syntaxique	Cohérence sémantique
Quatre	Restrictions portant sur l'utilisation des expressions PCL dans les cas particuliers de définitions de slots, et requêtes	
Trois	Définition d'une expression PCL, et conditions et propriétés faisant intervenir la définition d'une expression PCL	Sémantique associée aux concepts définis syntaxiquement au niveau TROIS, et sémantique des expressions PCL
Deux	Conditions et propriétés portant sur les concepts du modèle, définis au niveau UN	Sémantique associée aux concepts définis syntaxiquement au niveau DEUX
Un	Formes syntaxiques des définitions de facettes	Sémantique des définitions de facettes, restreints aux cas où la sémantique d'une expression n'est pas nécessaire
Zéro	Définitions générales des concepts de base du meta-modèle conceptuel: notions de classes non restreintes par des conditions	Sémantique des concepts de base du modèle conceptuel

2. Description par niveau de cohérence

Les differents niveaux du méta-modèle de Base de Connaissances sont décrits ci-dessous.

Pour chacun d'eux, on donnera dans un premier temps la liste des concepts décrits syntaxiquement, et dans un second temps une enumeration des objets dont la semantique est enoncee.

2.1. Le niveau de cohérence ZERO

a. Syntaxe

Ce niveau décrit syntaxiquement les concepts suivants:

- Définition de facette
- Définition de slot
- Définition de classe
- Définition de relation IS-A
- Slot propre Sj d'une classe Cli
- Slot hérité Sj d'une classe Cli
- Nom de métaclasse
- Le modèle conceptuel
- Classes dites "basic".
- Classes dites "non-basic"

b. Sémantique

Les concepts pour lesquels la sémantique est définie à ce niveau sont:

- Métaclasse
- Définition de classe:
  ensemble d'instances
  slot implicite de nom **identifier**
- Définition de relation IS-A
- Instance propre
- Instance spécialisée
- Définition de slot: fonction associée à chaque slot
- Modèle conceptuel

2.2. Le niveau de cohérence UN

a. Syntaxe

Ce niveau décrit syntaxiquement les concepts suivants:

- Graphe sans cycle défini par l'ensemble des relations IS-A.
- Forme de la définition de facette categ (categ, *value*).
- Slot de propriété
- Slot de contrainte
- Forme de la définition de facette card (card, m-n).
- Forme de la définition de facette comment (comment, *value*).
- Forme de la définition de facette presence (presence, *value*),
- Forme de la définition de facette reverse (reverse, *slotname*).

b. Sémantique

Les concepts pour lesquels la sémantique est définie à ce niveau sont:

- Facette (comment, value)
- Facette (presence, value):
  fonction partielle ou totale associée à chaque slot

2.3. Le niveau de cohérence DEUX

a. Syntaxe

Ce niveau décrit syntaxiquement les conditions et propriétés suivantes:

- Pour les Classes enumerated:
  définition de slot autorisée.
- Pour les Classes range:
  définitions de slot autorisées.
- Pour les Classes primitives:
  ensemble vide de définitions de slot.
- Pour la facette def: obligatoire pour tout slot.
- Pour la facette card: conditions d'utilisation
- Pour la facette presence: conditions d'utilisation.
- Pour la facette reverse: conditions d'utilisation.
- Pour la facette default: conditions d'utilisation.

b. Sémantique

Les concepts pour lesquels la sémantique est définie à ce niveau sont:

- Classes dites basic:
  définition précise de l'ensemble d'instances.
- Classes dites non-basic: précisions sur l'ensemble d'instances
  slots de propriété: fontionnels ou relationnesl.
  slots de contrainte

2.4. Le niveau de cohérence TROIS

Précondition:

A ce niveau, pour une Base de Connaissances particulière, on considère que tous les slots possedent une valeur (fixée) pour chaque instance de chaque classe.

a. Syntaxe

Ce niveau décrit syntaxiquement les propriétés, conditions, et concepts suivants:

- Définition syntaxique d'une expression PCL.
- Pour la facette def d'un slot:

  expression PCL syntaxiquement correcte.
- Pour la facette default d'un slot:

  valeur respectant le type de l'expression PCL syntaxiquement correcte.
- Pour la facette reverse d'un slot: conditions d'utilisation.

b. Sémantique

Les concepts pour lesquels la sémantique est définie à ce niveau sont:

- Facette (card, value):

  précisions sur l'ensemble d'arrivée de la fonction associee au slot contenant une telle facette.
- Classes dites non-basic:

  précisions sur la fonction associée à chaque slot d'une telle classe.
- Facette (reverse, value):

  précisions sur la fonction associée à chaque slot contenant une telle facette.
- Facette (default, value):

  précisions sur la valeur d'une telle facette.
- Sémantique associée à une expression PCL.

2.5. Le niveau de cohérence QUATRE

Precondition:

Il est à noter que la sémantique des expressions PCL est définie sur une Base de Connaissances particulière (fixée, statique) qui doit être syntaxiquement et sémantiquement cohérente avec le niveau TROIS.

a. Syntaxe

Ce niveau décrit syntaxiquement les restrictions des expressions PCL dans les cas suivants:

- facettes def des slots de propriété,
- facettes def des slots de contrainte,
- requetes.

b. Sémantique

Aucune sémantique particulière n'est associée à ce niveau, le niveau QUATRE ne définissant syntaxiquement que des restrictions sur l'utilisation des expressions PCL.

Contexte
de définition d'une expression

Annexe 3

1. Présentation du contexte de définition des expressions

Le langage PCL étant un langage de requêtes et d'expression de contraintes, il est possible, dès à présent, de déterminer certaines caractéristiques de ces expressions.

Ceci permettra de se faire une idée plus précise aussi bien du contexte dans lequel les expressions seront formulées, que de certains paramètres influençant la définition ou l'évaluation d'une expression.

Une première caractéristique du contexte dans lequel une expression peut être formulée est déjà apparue dans le chapitre 1 de la troisième partie.

En effet, toute expression de contrainte ou requête ne peut être définie que par rapport à une Base de Connaissances particulière (modèle conceptuel et ensemble d'instances).

Partant du contenu attendu de PCL (décrit dans la première partie), il est possible de formuler certaines caractéristiques des expressions, et de cette manière de pouvoir relever certains paramètres qui vont influencer soit la formulation, soit l'évaluation d'une expression :

- Une expression doit pouvoir contenir des variables libres.

  Dès lors, on peut concevoir l'importance d'une liste (ou table) reprenant chacune des variables.

  En effet, cette liste (ou table) devra vraisemblablement contenir une information concernant le type d'objets référencés par chaque variable pour permettre des vérifications syntaxiques.

  D'autre part, il est probable qu'une liste (ou table), contenant pour chaque variable la valeur qui lui est associée, puisse être un paramètre influençant l'évaluation de l'expression.

- une requête peut contenir un nom de classe ou de propriété (slot) d'une classe.

  De manière similaire au cas précédent, on peut concevoir l'importance d'une liste (ou table) reprenant chacune des classes apparaissant dans l'expression ou y étant simplement référencée.

  En effet, cette liste (ou table) devra également permettre certaines vérifications syntaxiques, et fournir des informations utiles pour la phase d'évaluation.

- La forme générale d'une expression étant censée désigner un ensemble d'éléments, il paraît important de distinguer deux types d'expressions :

  Une première catégorie regrouperait les expressions désignant un seul élément (et pas un ensemble d'un seul élément).

  Cette catégorie contiendrait au minimum toutes les contraintes. Une contrainte peut en effet être définie comme une expression à valeur booléenne.

  Remarquons qu'une telle requête (pour laquelle le résultat de l'évaluation est un seul élément) peut cependant fournir différentes valeurs, si elle est évaluée par exemple pour un autre jeu d'instanciation des variables.

  La seconde catégorie reprendrait toute expression pour laquelle le résultat de l'évaluation serait un ensemble de valeurs.

  L'évaluation d'une telle requête peut cependant fournir différents ensembles de valeurs, si elle est évaluée par exemple pour un autre jeu d'instanciation des variables.

Présentation intuitive des expressions du langage

Annexe 4

1. Introduction

Ce document presente de maniere intuitive la forme des expressions abstraites du langage PCL.

Cette présentation commence par une description très schématique du langage PCL, ainsi que de la manière dont sont construites les expressions.

Une seconde partie décrit les expressions de base du langage, tandis que la troisième comprend une analyse pour chaque opérateur abstrait. Cette analyse est établie selon le plan suivant :

Opérateur **OP** :

- arité de l'opérateur
- arguments

  pour chaque argument, une description de celui-ci indiquant de manière intuitive ses caractéristiques.
- syntaxe abstraite utilisée
- description de la sémantique associée à l'opérateur
- résultat de l'évaluation de l'expression

2. Description intuitive du langage

La description intuitive, et très schematique, du langage PCL (langage d'expressions, et de contraintes) consiste en une description des objets de base du langage, suivie d'une description des operateurs permettant de former des expressions plus complexes à partir d'expressions simples.

Expressions primitives :

1. constante
2. nom de classe
3 variable
4. nom de slot

Expressions composées :

1. $f(expr_1, \ldots, expr_n)$

où

- f est un opérateur abstrait
- n est l'arité de l'opérateur f
- $expr_i$ est une expression PCL ($\forall$ i : 1<=i<=n)

2. $(t_1, \ldots, t_n)$

une telle expression est appelée t-uple

où

- n >= 1
- ($\forall$ i : 1<=i<=n) : t_i est une expression PCL
- $t_i \neq t_j$ ($\forall$ i ≠ j)

3. $[c_1, \ldots, c_n]$

une telle expression est appelée ensemble explicite de valeurs

où

- n >= 0
- ($\forall$ i : 1<=i<=n) : c_i est une constante
- $\forall$ i , j : c_i et c_j sont de même type

3. Les expressions de base

Les expressions de base du langage PCL peuvent être classees en deux catégories :

- les expressions primitives.
- les expressions composées construites à partir d'expressions plus simples.

Les expressions primitives

1 constante

Chaque constante est une valeur simple qui peut être :

- un entier
- un réel
- un string : une chaîne de caractères

Une constante est donc chaque instance d'une classe dite basic.

2. nom de classe

Un nom de classe est considéré comme une expression primitive du langage ; ce nom de classe doit évidement correspondre à une classe du modèle conceptuel.

3. variable

Une variable est une expression primitive du langage ; elle peut être utilisée pour remplacer une expression PCL de type quelconque.

Une variable peut donc désigner une expression primitive, aussi bien qu'une expression composée de n'importe quel type.

4. nom de slot

De manière analogue à un nom de classe, un nom de slot est considéré comme une expression primitive du langage ; ce nom de slot doit évidement correspondre à un slot défini pour une classe du modèle conceptuel.

Les expressions composées

1. $f(expr_1, \ldots, expr_n)$

où

- f est un opérateur abstrait
- n est l'arité de l'opérateur f
- $expr_i$ est une expression PCL ($\forall$ i : 1<=i<=n)

Les différentes classes d'opérateurs sont les suivantes :

- Opérateur associant un slot à une classe
- Opérateur formant un ensemble
- Opérateur posant une condition sur un objet
- Operateurs logiques :
  négation, conjonction, disjonction, opérateurs de quantification
- Opérateurs de comparaison
  égalité, non-égalité, plus grand, strictement plus grand, plus petit, strictement plus petit,
- Opérateurs ensemblistes
  appartenance, inclusion
- Opérateurs définissant des fonctions aggrégées
  moyenne, minimum, maximum, somme, cardinal
- Opérateurs arithmétiques
  addition, soustraction, multiplication, division

Une description plus précise d'une telle expression est donnée pour chaque opérateur abstrait f (Voir **4**).

2. $(t_1, \ldots, t_n)$

une telle expression est appelée t-uple

où

- n >= 1
- ($\forall$ i : 1<=i<=n) : t_i est une expression PCL

t_i ne peut être une variable sur laquelle aucune condition n'est posée (la condition la plus simple devant indiquer la classe référenciée par la variable).

- $t_i \neq t_j$ ($\forall$ i $\neq$ j)

3. [c_1, ... ,c_n]

une telle expression est appelée ensemble explicite de valeurs

où

- n >= 0
- ($\forall$ i : 1<=i<=n) : c_i est une constante
- $\forall$ i , j : c_i et c_j sont de même type

4. Les opérateurs

Chaque operateur est décrit par son arité, la nature de ses arguments, sa syntaxe abstraite, une description de sa sématique, ansi que la nature du résultat.

4.1. Opérateur associant un slot à une classe

A. Opérateur ISofClass

arité : 2

arguments :

1. un nom de slot
2. un nom de classe, une variable désignant une classe, ou une expression PCL contenant l'opérateur ISofClass.

syntaxe abstraite : ISofClass($expr_1$, $expr_2$)

description : le slot désigné par le premier argument appartient à la définition de classe désignée par le second.

résultat : la valeur du slot désigné par le premier argument

4.2. Opérateur formant un ensemble

A. Opérateur SET

arité : 1

argument :

1. une expression PCL (nom de classe ou variable sur laquelle porte éventuellement une condition, tuple)

syntaxe abstraite : SET($expr_1$)

description : cette expression désigne l'ensemble des éléments satisfaisant l'expression $expr_1$.

résultat : l'ensemble formé par les éléments de l'évaluation de l'expression PCL $expr_1$

4.3. Opérateur posant une condition sur un objet

A. Opérateur COND

arité : 2

arguments :

1. nom de classe dite non-basic, variable désignant un nom de classe dite non-basic
2. expression PCL de type boolean

syntaxe abstraite : COND($expr_1$, $expr_2$)

description : tout élément de la classe référencée par le premier argument doit satisfaire l'expression booléenne énoncée par le deuxième argument

résultat : tout élément de la classe référencée par le premier argument et satisfaisant l'expression booléenne $expr_2$.

4.4. Opérateurs logiques

A. Opérateur NOT

arité : 1

argument :

1. une expression de type boolean.

syntaxe abstraite : NOT($expr_1$)

description : négation logique de l'expression $expr_1$

résultat : la valeur booléenne qui est la négation logique de la valeur booléenne désignée par l'expression $expr_1$.

B. Opérateur AND

arité : 2

arguments :

1. une expression de type boolean
2. une expression de type boolean

syntaxe abstraite : AND($expr_1$, $expr_2$)

description : conjonction logique des deux expressions $expr_1$ et $expr_2$

résultat : valeur booléenne de la conjonction logique des deux expressions $expr_1$ et $expr_2$

C. Opérateur OR

arité : 2

arguments :

1. une expression de type boolean
2. une expression de type boolean

syntaxe abstraite : OR($expr_1$, $expr_2$)

description : disjonction logique des deux expressions $expr_1$ et $expr_2$

résultat : valeur booléenne de la disjonction logique des deux expressions $expr_1$ et $expr_2$

D. Opérateurs de quantification

D1. Opérateur EXIST-Member

arité : 3

arguments :

1. variable
2. expression désignant un ensemble
3. expression de type boolean

syntaxe abstraite : EXIST-Member($expr_1$, $expr_2$, $expr_3$)

description : expression logique ($\exists$ **$expr_1$** $\in$ **$expr_2$** : **$expr_3$**), où expr1 est une variable désignant une valeur simple.

résultat : valeur booléenne de l'expression logique

D2. Opérateur EXIST-Included

arité : 3

arguments :

1. variable
2. expression désignant un ensemble
3. expression de type boolean

syntaxe abstraite : EXIST-Included($expr_1$, $expr_2$, $expr_3$)

description : expression logique ($\exists$ **$expr_1$ C $expr_2$: $expr_3$**), où expr1 est une variable désignant un ensemble d'éléments.

résultat : valeur booléenne de l'expression logique

D3. Opérateur EXIST-ISIN

arité : 3

arguments :

1. variable
2. expression désignant un ensemble
3. expression de type boolean

syntaxe abstraite : EXIST-ISIN($expr_1$, $expr_2$, $expr_3$)

description : expression logique ($\exists$ **$expr_1 \in expr_2$: $expr_3$**), où $expr_1$ est une variable désignant un élément de $expr_2$, et où l'appartenance de $expr_1$ à $expr_2$ est limitée à une vérification de type (appartenance à un type restreint).

résultat : valeur booléenne de l'expression logique

D4. Opérateur FORALL-Member

arité : 3

arguments :

1. variable
2. expression désignant un ensemble
3. expression de type boolean

syntaxe abstraite : FORALL-Member($expr_1$, $expr_2$, $expr_3$)

description : expression logique ($\forall$ **$expr_1 \in expr_2$: $expr_3$**), où expr1 est une variable désignant une valeur simple.

résultat : valeur booléenne de l'expression logique

D5. Opérateur FORALL-Included

arité : 3

arguments :

1. variable
2. expression désignant un ensemble
3. expression de type boolean

syntaxe abstraite : FORALL-Included($expr_1$, $expr_2$, $expr_3$)

description : expression logique (**$\forall$ $expr_1$ C $expr_2$: $expr_3$**), où expr1 est une variable désignant un ensemble d'éléments.

résultat : valeur booléenne de l'expression logique

D3. Opérateur FORALL-ISIN

arité : 3

arguments :

1. variable
2. expression désignant un ensemble
3. expression de type boolean

syntaxe abstraite : FORALL-ISIN($expr_1$, $expr_2$, $expr_3$)

description : expression logique (**$\forall$ $expr_1$ $\in$ $expr_2$: $expr_3$**), ou $expr_1$ est une variable désignant un élément de $expr_2$, et où l'appartenance de $expr_1$ à $expr_2$ est limitée à une vérification de type (appartenance à un type restreint).

résultat : valeur booléenne de l'expression logique

4.5. Opérateurs de comparaison

A. Opérateurs GT, GE, ST, SE

arité : 2

arguments :

1. expression de type integer, real, ou string
2. expression de type integer, real, ou string

syntaxe abstraite : respectivement

GT($expr_1$, $expr_2$),
GE($expr_1$, $expr_2$),
ST($expr_1$, $expr_2$),
SE($expr_1$, $expr_2$).

description : respectivement expression logique
(**$expr_1$ > $expr_2$**),
(**$expr_1$ >= $expr_2$**),
(**$expr_1$ < $expr_2$**),
(**$expr_1$ <= $expr_2$**).

résultat : valeur de l'expression logique

B. Opérateurs EQ, NE

arité : 2

arguments :

1. expression désignant un seul élément
2. expression désignant un seul élément

syntaxe abstraite : respectivement

EQ($expr_1$, $expr_2$),
NE($expr_1$, $expr_2$),

description : respectivement expression logique
(**$expr_1$ = $expr_2$**),
(**$expr_1$ ≠ $expr_2$**),

résultat : valeur de l'expression logique

4.6. Opérateurs ensemblistes

A. Opérateur Member

arité : 2

arguments :

1. expression désignant un seul élément
2. expression désignant un ensemble d'éléments

syntaxe abstraite : Member($expr_1$, $expr_2$)

description : expression logique (**$expr_1 \in expr_2$**), où expr1 désigne une valeur simple.

résultat : valeur booléenne de l'expression logique

B. Opérateurs Included

arité : 2

arguments :

1. expression désignant un ensemble d'éléments
2. expression désignant un ensemble d'éléments

syntaxe abstraite : Included($expr_1$, $expr_2$)

description : expression logique (**$expr_1$ C $expr_2$**), où expr1 désigne un ensemble d'éléments.

résultat : valeur booléenne de l'expression logique

C. Opérateur ISIN

arité : 2

arguments :

1. expression désignant un seul élément
2. expression désignant un ensemble d'éléments

syntaxe abstraite : ISIN($expr_1$, $expr_2$)

description : expression logique (**$expr_1 \in expr_2$**), où expr1 désigne un élément, et où l'appartenance de expr1 à expr2 est limitée à une vérification de type (appartenance à un type restreint).

résultat : valeur booléenne de l'expression logique

4.7. Opérateurs définissant des fonctions agrégées

A. Opérateur AVG

arité : 1

argument :

1. expression désignant un ensemble d'éléments

syntaxe abstraite : AVG($expr_1$)

description : expression représentant la moyenne des éléments de $expr_1$

résultat : valeur (réelle) de la moyenne des éléments de $expr_1$.

B. Opérateurs MIN, MAX

arité : 1

argument :

1. expression désignant un ensemble d'éléments

syntaxe abstraite :

description : expression du minimum (maximum) des éléments de l'ensemble $expr_1$

résultat : valeur du minimum (maximum) des éléments de l'ensemble $expr_1$.

C. Opérateur SUM

arité : 1

argument :

1. expression désignant un ensemble d'éléments

syntaxe abstraite : SUM($expr_1$)

description : expression de la somme des éléments de l'ensemble $expr_1$

résultat : valeur (entière ou réelle) de la somme des éléments de l'ensemble $expr_1$.

D. Opérateur COUNT

arité : 1

argument :

1. expression désignant un ensemble d'éléments

syntaxe abstraite : COUNT($expr_1$)

description : expression de cardinal de l'ensemble $expr_1$

résultat : valeur (entière) du cardinal de l'ensemble $expr_1$.

4.8. Opérateurs arithmétiques

A. Opérateurs PLUS, MINUS, TIMES, DIV

arité : 2

arguments :

1. expression de type integer ou real
2. expression de type integer ou real

syntaxe abstraite : respectivement

PLUS($expr_1$, $expr_2$),
MINUS($expr_1$, $expr_2$),
TIMES($expr_1$, $expr_2$),
DIV($expr_1$, $expr_2$).

description : respectivement l'expression arithmétique

$expr_1$ + $expr_2$,
$expr_1$ - $expr_2$,
$expr_1$ * $expr_2$,
$expr_1$ / $expr_2$,

résultat : valeur (entière ou réelle) respectivement de la somme, différence, multiplication, division des expressions $expr_1$ et $expr_2$

Grammaire attribuée définissant la syntaxe abstraite

Annexe 5

1. Introduction

Ce document présente la définition syntaxique de la grammaire abstraite du langage PCL.

Une première partie définit le type d'une expression de base.

Le second point constitue la description de la grammaire attribuée définissant la correction syntaxique d'une expression, tandis que le troisième point constitue la définition de la grammaire attribuée pour chaque forme d'expression de base.

2. Type d'une expression

Toute expression du langage est d'un type déterminé, qui est l'un des types suivants :

- integer
- real
- string
- *classname*, où *classname* est un nom d'une classe appartenant au modèle conceptuel
- tuple
- set(*X*), où *X* est l'un des cinq types précédents
- boolean

Avant de définir la correction syntaxique d'une expression, il est donc nécessaire de définir le type de chaque expression, ainsi que les règles de compatibilité de types résultant de l'usage d'opérateurs.

2.1. Expressions de base

Pour chaque expression de base du langage, on définit le type de l'expression.

2.1.1. Expressions primitives

Constantes :

Le type d'une constante, notée cst, est :

- **integer** :

  si cst est une constante entière

- **real** :

si cst est une constante réelle

- **string** :

  si cst est une constante représentant une chaîne de caractères

Noms de classe :

Le type d'une expression primitive réduite à un nom de classe *classname* est :

- le type *classname* si la classe est dite non-basic.
- **integer, real, string** si la classe est une classe dite basic.

  Les classes enumerated sont de type string,

  Les classes range sont de type integer ou real selon la valeur du slot extension,

  Les classes primitives integer, integer(m,n) sont de type integer,

  Les classes primitives string, string(m,n) sont de type string,

  La classe primitive real est de type real.

Noms de slot :

Le type d'une expression primitive réduite à un nom de slot *slotname* est le type de l'expression PCL donnée par la facette def du slot.

Ce type peut être :

- integer, real, string,
- classname,
- set(integer), set(real), set(string),
- set(classname).

Variables :

Le type d'une variable est l'un des types suivants, en fonction du type de l'objet référencé par la variable :

- integer, real, string,

- classname,
- tuple,
- set(integer), set(real), set(string),
- set(classname),
- set(tuple).

2.1.2. Expressions composées

Ensemble explicite de valeurs :

Le type d'un ensemble explicite de valeurs est, de part la définition d'un tel ensemble, **set(*X*)** où *X* est le type d'un élément de l'ensemble.

Rappelons qu'un ensemble explicite de valeurs contient des éléments tous de même type.

Tuple :

Le type d'un tupe est **tuple** .

$f(expr_1, \ldots\ldots, expr_n)$:

Le type d'une telle expression est donné par une table de compatibilité de types, pour chaque opérateur, au point suivant.

2.2. Compatibilité de type et type résultant des opérateurs

Pour chaque opérateur, on définit une table contenant :

- le type de chaque opérande intervenant dans l'expression,
- la correction du point de vue de la compatibilité des types des opérandes, cette valeur étant donnée par une fonction ayant comme argument les types des opérandes et définie de la manière suivante :

  arité : n + 1

  arguments :

  1. **OP**
  2. type($expr_1$), désignant le type de l'opérande $expr_1$

.....

(n+1). type($expr_n$), désignant le type de l'opérande $expr_{n+1}$

notation : **comp(OP**, type(expr1), ... , type($expr_n$) **)**

- et, le type résultant de l'expression en fonction des types des opérandes, ce type étant également donné par une fonction ayant comme argument les types des opérandes et définie de la manière suivante :

  arité : n + 1

  arguments :

  1. **OP**

  2. type($expr_1$), désignant le type de l'opérande $expr_1$

  .....

  (n+1). type($expr_n$), désignant le type de l'opérande $expr_{n+1}$

  notation : **result-type(OP**, type(expr1),..... , type($expr_n$) **)**

Notation utilisée

Soit l'opérateur **OP** :

L'opérateur **OP** est d'arité n.

- les n premières colonnes sont intitulées **$expr_i$** , où **$expr_i$** donne le type de l'opérande $expr_i$. (1 <= i <= n)
- la colonne intitulée **comp** donne la valeur de la fonction de compatibilité.
- la colonne intitulée **result-type** donne la valeur de la fonction donnant le type résultant.

2.2.1. Opérateur associant un slot à une classe

A. ISofClass($expr_1$, $expr_2$)

expr1	expr2	comp	result type
class1	set(class2),class2	correct	class1
integer	set(class2),class2	correct	integer
real	set(class2),class2	correct	real
string	set(class2),class2	correct	string
set(class1)	set(class2),class2	correct	set(class1)
set(integer)	set(class2),class2	correct	set(integer)
set(real)	set(class2),class2	correct	set(real)
set(string)	set(class2),class2	correct	set(string)
others	others	not correct	undetermined

où

- class1, class2 sont des noms de classes dites non-basic,
- class1 peut être distinct de class2.

2.2.2. Opérateur formant un ensemble

A. SET($expr_1$)

expr1	comp	result type
tuple	correct	set(tuple)
class1	correct	set(class1)
integer	correct	set(integer)
real	correct	set(real)
string	correct	set(string)
others	not correct	undetermined

où

- class1 est un nom de classe dite non-basic.

2.2.3. Opérateur posant une condition sur un objet

A. COND($expr_1$,$expr_2$)

expr1	expr2	comp	result type
class1	boolean	correct	class1
integer	boolean	correct	integer
real	boolean	correct	real
string	boolean	correct	string
others	others	not correct	undetermined

où

- class1 est un nom de classe dite non-basic.

2.2.4. Opérateurs logiques

A. NOT($expr_1$)

expr1	comp	result type
boolean	correct	boolean
others	not correct	undetermined

B. AND($expr_1$, $expr_2$), OR($expr_1$, $expr_2$)

expr1	expr2	comp	result type
boolean	boolean	correct	boolean
others	others	not correct	undetermined

C. EXIST-Included($expr_1$, $expr_2$, $expr_3$), FORALL-Included($expr_1$, $expr_2$, $expr_3$)

expr1	expr2	expr3	comp	result type
set(tuple)	set(tuple)	boolean	correct	boolean
set(class1)	set(class1)	boolean	correct	boolean
set(integer)	set(integer)	boolean	correct	boolean
set(real)	set(real)	boolean	correct	boolean
set(string)	set(string)	boolean	correct	boolean
others	others	others	not correct	undetermined

où

- class1 est un nom de classe dite non-basic.

D. EXIST-Member($expr_1$, $expr_2$, $expr_3$), FORALL-Member($expr_1$, $expr_2$, $expr_3$)

expr1	expr2	expr3	comp	result type
tuple	set(tuple)	boolean	correct	boolean
class1	set(class1)	boolean	correct	boolean
integer	set(integer)	boolean	correct	boolean
real	set(real)	boolean	correct	boolean
string	set(string)	boolean	correct	boolean
others	others	others	not correct	undetermined

où

- class1 est un nom de classe dite non-basic.

E. EXIST-ISIN($expr_1$, $expr_2$, $expr_3$), FORALL-ISIN($expr_1$, $expr_2$, $expr_3$)

expr1	expr2	expr3	comp	result type
class1	class1	boolean	correct	boolean
integer	integer	boolean	correct	boolean
real	real	boolean	correct	boolean
string	string	boolean	correct	boolean
set(class1)	set(class1)	boolean	correct	boolean
set(integer)	set(integer)	boolean	correct	boolean
set(real)	set(real)	boolean	correct	boolean
set(string)	set(string)	boolean	correct	boolean
others	others	others	not correct	undetermined

où

- class1 est un nom de classe dite non-basic.

2.2.5. Opérateurs de comparaison

A. GT($expr_1$, $expr_2$), GE($expr_1$, $expr_2$), ST($expr_1$, $expr_2$), SE($expr_1$, $expr_2$)

expr1	expr2	comp	result type
integer	integer	correct	boolean
integer	real	correct	boolean
real	integer	correct	boolean
real	real	correct	boolean
string	string	correct	boolean
others	others	not correct	undetermined

B. EQ($expr_1$, $expr_2$), NE($expr_1$, $expr_2$)

expr1	expr2	comp	result type
tuple	tuple	correct	boolean
class1	class2	correct	boolean
integer	integer	correct	boolean
real	real	correct	boolean
string	string	correct	boolean
others	others	not correct	undetermined

où

- class1, class2 sont des noms de classes dites non-basic,
- class1 peut être distinct de class2.

2.2.6. Opérateurs ensemblistes

A. Member($expr_1$, $expr_2$)

expr1	expr2	comp	result type
tuple	set(tuple)	correct	boolean
class1	set(class2)	correct	boolean
integer	set(integer)	correct	boolean
real	set(real)	correct	boolean
string	set(string)	correct	boolean
others	others	not correct	undetermined

où

- class1, class2 sont des noms de classes dites non-basic,
- class1 peut être :
    - le même que class2
    - le nom d'une sous-classe de class2

B. Included($expr_1$, $expr_2$), EqSet($expr_1$, $expr_2$)

expr1	expr2	comp	result type
set(tuple)	set(tuple)	correct	boolean
set(class1)	set(class2)	correct	boolean
set(integer)	set(integer)	correct	boolean
set(real)	set(real)	correct	boolean
set(string)	set(string)	correct	boolean
others	others	not correct	undetermined

où

- class1, class2 sont des noms de classes dites non-basic,
- class1 peut être :
    - le même que class2
    - le nom d'une sous-classe de class2

C. ISIN($expr_1$, $expr_2$)

expr1	expr2	comp	result type
class1	class2	correct	boolean
integer	integer	correct	boolean
real	real	correct	boolean
string	string	correct	boolean
set(class1)	set(class2)	correct	boolean
set(integer)	set(integer)	correct	boolean
set(real)	set(real)	correct	boolean
set(string)	set(string)	correct	boolean
others	others	not correct	undetermined

où

- class1, class2 sont des noms de classes dites non-basic,
- class1 peut être :
    - le même que class2
    - le nom d'une sous-classe de class2

2.2.7. Opérateurs définissant des fonctions agrégées

A. AVG($expr_1$)

expr1	comp	result type
set(integer)	correct	real
set(real)	correct	real
others	not correct	undetermined

B. MIN($expr_1$), MAX($expr_1$)

expr1	comp	result type
set(integer)	correct	integer
set(real)	correct	real
set(string)	correct	string
others	not correct	undetermined

C. SUM($expr_1$)

expr1	comp	result type
set(integer)	correct	integer
set(real)	correct	real
others	not correct	undetermined

D. COUNT($expr_1$)

expr1	comp	result type
set(tuple)	correct	integer
set(class1)	correct	integer
set(integer)	correct	integer
set(real)	correct	integer
set(string)	correct	integer
others	not correct	undetermined

où

- class1 est un nom de classe dite non-basic,

2.2.8. Opérateurs arithmétiques

A. PLUS($expr_1$, $expr_2$), MINUS($expr_1$, $expr_2$), TIMES($expr_1$, $expr_2$)

expr1	expr2	comp	result type
integer	integer	correct	integer
integer	real	correct	real
real	integer	correct	real
real	real	correct	real
others	others	not correct	undetermined

B. DIV($expr_1$, $expr_2$)

expr1	expr2	comp	result type
integer	integer	correct	real
integer	real	correct	real
real	integer	correct	real
real	real	correct	real
others	others	not correct	undetermined

3. Description de la grammaire

La description de la grammaire consiste en la définition de chacun de ses attributs.

Un attribut de la grammaire est une fonction définie par :

- le nom de la fonction (par convention, devenu le nom de l'attribut),
- l'ensemble de départ
- l'ensemble d'arrivée

Un dernier point décrira de manière succincte la sémantique de chaque attribut.

3.1. Attributs utilisés

1. Attribut "synt"

synt : Expr ----------> C

exp ~~~~~~~~> synt(exp)

2. Attribut "type"

type : Expr ----------> Ty

exp ~~~~~~~~> type(exp)

3. Attribut "inh-ref"

inh-ref : Expr ----------> ListClassName

exp ~~~~~~~~> inh-ref(exp)

4. Attribut "ref"

ref : Expr ----------> ListClassName

exp ~~~~~~~~> ref(exp)

5. Attribut "inh-tabVar"

inh-tabVar : Expr ----------> ListVariable

exp ~~~~~~~~> inh-tabVar(exp)

7. Attribut "tabVar"

tabVar : Expr ----------> ListVariable

exp ~~~~~~~~> tabVar(exp)

8. Attribut "object"

object : Expr ----------> Obj

exp ~~~~~~~~> object(exp)

9. Attribut "class-refer"

class-refer : Expr ----------> Clname

exp ~~~~~~~~> class-refer(exp)

Où les différents ensembles sont :

- Expr = { expressions }
- C = { correct, not correct }
- Ty = { tuple, set(tuple), classname, set(classname), integer, set(integer), real, set(real), string, set(string), boolean, undet[ermined] }
- Obj = { cst, class, var, exp, cond }
- ListClassName = { [x | y] tel que x est un nom de classe et y est un élément de ListClassName }
- ListVariable = { ‹ var, typevar, classref › tel que var est une variable, typevar est le type de la variable var, et classref est la classe référencée par la variable var }
- Clname = { classname} + {no-value}

3.2. Sémantique des attributs

1. Attribut "synt"

La fonction **synt** associe à chaque expression exp une valeur synt(exp) qui est :

synt(exp) =

correct si exp est une expression qui satisfait les règles syntaxiques de la syntaxe abstraite, par rapport au modèle conceptuel considéré.

not correct sinon.

2. Attribut "type"

La fonction **type** associe à chaque expression exp le type de cette expression.

3. Attributs "inh-ref" et "ref"

Rappelons que la liste des classes, pouvant être référencées par un nom de slot, est un paramètre qui concerne la syntaxe d'une expression.

Cette liste de classes peut être divisée en deux listes distinctes de noms de classes :

- la première contient tous les noms de classes qui sont "hérités" d'expressions plus générales (tous les noms de classes qui sont apparus dans l'expression éventuelle contenant l'expression analysée).
- la seconde contient tous les noms de classes qui apparaissent explicitement dans l'expression analysée.

La fonction **inh-ref** associe à chaque expression exp la liste des noms de classes qui sont "hérités" de l'expresion éventuelle contenant exp.

La fonction **ref** associe à chaque expression exp la liste des noms de classes qui apparaissent explicitement dans exp.

Les deux listes sont des listes ordonnées de type Last in - First out".

4. Attributs "inh-tabVar" et "tabVar"

Rappelons que l'ensemble des variables pouvant apparaître dans l'expression est un paramètre qui concerne la syntaxe de cette expression.

Cet ensemble de variables peut être divisée en deux ensembles distincts de variables :

- le premier contient toutes les variables qui sont "héritées" d'expressions plus générales (toutes les variables qui sont apparues dans l'expression éventuelle contenant l'expression analysée).
- le second contient toutes les variables qui apparaissent explicitement dans l'expression analysée.

La fonction **inh-tabVar** associe à chaque expression exp l'ensemble des variables qui sont "héritées" de l'expresion éventuelle contenant exp.

La fonction **tabVar** associe à chaque expression exp l'ensemble des variables qui apparaissent explicitement dans exp.

Les deux ensembles sont des ensembles non-ordonnes.

5. Attribut "object"

La fonction **object** associe à chaque expression exp la sorte d'objet désigné par cette expression :

- cst pour une expression constante
- class pour un nom de classe
- var pour une variable
- cond pour une expression booléenne
- exp pour tous les autres types d'expressions

6. Attribut "class-refer"

La fonction **class-refer** associe à chaque expression exp une valeur class-refer(exp) qui est :

class-refer(exp) =

le nom de classe d'une classe dite basic ou non-basic référencée par l'expression expr,

no-value sinon (si exp ne référence pas une classe).

4. Grammaire attribuée

La correction syntaxique d'une expression est constituée de la définition de la correction syntaxique de chaque expression de base.

De cette manière, pour chaque forme particulière d'une expression, on définit la valeur des attributs de la grammaire attribuée.

Un dernier point précisera la correction syntaxique d'une expression dans les cas particuliers de requête ou d'expression définissant un slot.

4.1. Correction syntaxique d'une expression

Les pages suivantes constituent la définition de la correction syntaxique d'une expression selon la forme de l'expression.

4.1.1. Expression de base

1. expr = constante

Attribut synt
synt(expr) = correct

Attribut type
type(expr) = selon la valeur de constante :

integer	si constante est un entier
real	si constante est un réel
string	si constante est un string

Attributs inh-ref et ref
inh-ref(constante) = inh-ref(expr)
ref(expr) = [ ]

Attributs inh-tabVar et tabVar
inh-tabVar(constante) = inh-tabVar(expr)
tabVar(expr) = { }

Attribut object
object(expr) = cst

Attribut class-refer
class-refer(expr) = no-value

2. expr = classname

Attribut synt
synt(expr) = (classname ∈ modèle conceptuel)
& (classname ∈ métaclasse MCi)

Attribut type
type(expr) = type(classname)

Attributs inh-ref et ref
inh-ref(classname) = inh-ref(expr)
ref(expr) = [classname]

Attributs inh-tabVar et tabVar
inh-tabVar(classname) = inh-tabVar(expr)
tabVar(expr) = { }

Attribut object
object(expr) = class

Attribut class-refer
class-refer(expr) = classname

Note :
type(classname) = selon la métaclasse MCi :

integer	si MCi est une des suivantes : - primitive, et classname = integer, integer(m,n) - range, si classname contient une facette (def,integer) pour le slot type .
real	si MCi est une des suivantes : - primitive, et classname = real, - range, si classname contient une facette (def,real) pour le slot type .
string	si MCi est une des suivantes : - primitive, et classname = string, string(m,n), - enumerated.
classname	si classname est dite non-basic
undet[ermined]	sinon

3. expr = variable

Attribut synt
  synt(expr) = ∃ ‹variable, typevar, Cl› ∈ inh-tabVar(expr)
    où typevar = type(expr), Cl = class-refer(expr)

Attribut type
  type(variable) = type(expr)

Attributs inh-ref et ref
  inh-ref(variable) = inh-ref(expr)
  ref(expr) = [ ]

Attributs inh-tabVar et tabVar
  inh-tabVar(variable) = inh-tabVar(expr)
  tabVar(expr) = { }

Attribut object
  object(expr) = var

Attribut class-refer
  class-refer(variable) = class-refer(expr)

4. expr = slotname

Attribut synt

synt(expr) =

(slotname ∈ modèle conceptuel)

& ∃ t_i ∈ inh-ref(expr) où t_i est le premier tel que :

(slotname C classe t_i)

& (slotname possède une facette (def,slot-expr))

Attribut type

type(expr) = type(slot-expr)

Attributs inh-ref et ref

inh-ref(slotname) = inh-ref(expr)

ref(expr) = [class-refer(slot-expr)]

Attributs inh-tabVar et tabVar

inh-tabVar(slotname) = inh-tabVar(expr)

tabVar(expr) = { }

Attribut object

object(expr) = exp

Attribut class-refer

class-refer(expr) = class-refer(slot-expr)

5. expr = (t1, ..., tn)

Attribut synt

synt(expr) =

$synt(t_1)$ & ..... & $synt(t_n)$

& ($\forall$ i : $object(t_i) \neq$ var)

& ($\forall$ i : $type(t_i) \neq$ boolean)

Attribut type

type(expr) = tuple

Attributs inh-ref et ref

$inh\text{-}ref(t_1)$ = [$ref(t_1)$ | $inh\text{-}ref(expr)$]]]

$\forall$ i : 1 < i <= n :

$inh\text{-}ref(t_i)$ = [$ref(t_{i-1})$ | [.....| [$ref(t_1)$ | $inh\text{-}ref(expr)$]]]

ref(expr) = [$ref(t_n)$ | [$ref(t_{n-1})$ | [..... | $ref(t_1)$]]]

Attributs inh-tabVar et tabVar

$inh\text{-}tabVar(t_1)$ = [$tabVar(t_1)$ | $inh\text{-}tabVar(expr)$]]]

$\forall$ i : 1 < i <= n :

$inh\text{-}tabVar(t_i)$ = $inh\text{-}tabVar(expr)$ U $tabVar(t_{i-1})$ U...U $tabVar(t_1)$

tabVar(expr) = $tabVar(t_n)$ U $tabVar(t_{n-1})$ U ..... U $tabVar(t_1)$

Attribut object

object(expr) = exp

Attribut class-refer

class-refer(expr) = no-value

6. expr = [c1, ..., cn]

Attribut synt

synt(expr) =
synt(c_1) & ..... & synt(c_n)
& ((∀ i : type(c_i) = integer)
ou (∀ i : type(c_i) = real)
ou (∀ i : type(c_i) = string)
)
& (∀ i : object(c_i) = cst)

Attribut type

type(expr) = set(type(c_1))

Attributs inh-ref et ref

inh-ref(c_1) = [ref(c_1) | inh-ref(expr)]]]
∀ i : 1 < i<= n :
inh-ref(c_i) = inh-ref(expr)
ref(expr) = [ref(c_n) | [ref(c_{n-1}) | [..... | ref(c_1)]]]

Attributs inh-tabVar et tabVar

inh-tabVar(c_1) = [tabVar(c_1) | inh-tabVar(expr)]]]
∀ i : 1 < i<= n :
inh-tabVar(c_i) = inh-tabVar(expr)
tabVar(expr) = tabVar(c_n) U tabVar(c_{n-1}) U ..... U tabVar(c_1)

Attribut object

object(expr) = exp

Attribut class-refer

class-refer(expr) = no-value

4.1.2. Expression contenant un opérateur

4.1.2.1. Opérateur associant un slot à une classe

1. expr = ISofClass(slotname, classname)

Attribut synt
synt(expr) =
synt(slotname) & synt(classname)
& comp(ISofClass,type(slotname),type(classname))
& object(classname) = class
& (slotname C classname class)

Attribut type
type(expr) =result-type (ISofClass,type(slotname),type(classname))

Attributs inh-ref et ref
inh-ref(slotname) = [classname | inh-ref(expr)]
inh-ref(classname) = [classname | inh-ref(expr)]
ref(expr) = [ref(classname) | ref(slotname)]

Attributs inh-tabVar et tabVar
inh-tabVar(slotname) = inh-tabVar(expr)
inh-tabVar(classname) = inh-tabVar(expr)
tabVar(expr) = tabVar(classname) U tabVar(slotname)

Attribut object
object(expr) = exp

Attribut class-refer
class-refer(expr) = class-refer(slotname)

2. expr = ISofClass(slotname, variable)

Attribut synt

synt(expr) =
synt(slotname) & synt(variable)
& comp(ISofClass,type(slotname),type(variable))
& (object(variable) = var)
& (∃ ‹variable, typevar, Clref› ∈ inh-tabVar(expr))
& (slotname C classe Clref)

Attribut type

type(expr) = result-type(ISofClass,type(slotname),type(variable))
type(variable) = typevar

Attributs inh-ref et ref

inh-ref(slotname) = [Clref | inh-ref(expr)]
inh-ref(variable) = [Clref | inh-ref(expr)]
ref(expr) = [ref(variable) | ref(slotname)]

Attributs inh-tabVar et tabVar

inh-tabVar(slotname) = inh-tabVar(expr)
inh-tabVar(variable) = inh-tabVar(expr)
tabVar(expr) = tabVar(variable) U tabVar(slotname)

Attribut object

object(expr) = exp

Attribut class-refer

class-refer(expr) = class-refer(slotname)
class-refer(variable) = Clref

3. expr = ISofClass(slotname, expr2)

Attribut synt

synt(expr) =
synt(slotname) & synt($expr_2$)
& comp(ISofClass,type(slotname),type($expr_2$))
& (slotname C class-refer($expr_2$))

Attribut type

type(expr) = result-type (ISofClass, type(slotname), type($expr_2$))

Attributs inh-ref et ref

inh-ref(slotname) = [class-refer($expr_2$) | inh-ref(expr)]
inh-ref($expr_2$) = [class-refer($expr_2$) | inh-ref(expr)]
ref(expr) = [ref(slotname) | ref($expr_2$)]

Attributs inh-tabVar et tabVar

inh-tabVar(slotname) = inh-tabVar(expr)
inh-tabVar($expr_2$) = inh-tabVar(expr)
tabVar(expr) = tabVar($expr_2$) U tabVar(slotname)

Attribut object

object(expr) = exp

Attribut class-refer

class-refer(expr) = class-refer(slotname)

4.1.2.2. Opérateur formant un ensemble

1. expr = SET(expr1)

ou

$expr_1$ n'est pas une variable

Attribut synt
synt(expr) =
synt($expr_1$)
& comp(SET,type($expr_1$))
& (object($expr_1$) ≠ var)

Attribut type
type(expr) = result-type(SET, type($expr_1$))

Attributs inh-ref et ref
inh-ref($expr_1$) = inh-ref(expr)
ref(expr) = ref($expr_1$)

Attributs inh-tabVar et tabVar
inh-tabVar($expr_1$) = inh-tabVar(expr)
tabVar(expr) = tabVar($expr_1$)

Attribut object
object(expr) = exp

Attribut class-refer
class-refer(expr) = class-refer($expr_1$)

2. expr = SET(expr1)

ou

$expr_1$ est une variable

Attribut synt

synt(expr) =
synt($expr_1$)
& comp(SET,type($expr_1$))
& (object($expr_1$) = var)
& (∃ ⟨$expr_1$, typevar, Clref⟩ ∈ inh-tabVar(expr))

Attribut type

type(expr) = result-type(SET, type($expr_1$))
type($expr_1$) =
typevar si ∃ ⟨$expr_1$, typevar, Clref⟩ ∈ inh-tabVar(expr)
undet sinon

Attributs inh-ref et ref

inh-ref($expr_1$) = inh-ref(expr)
ref(expr) = ref($expr_1$) = [ ]

Attributs inh-tabVar et tabVar

inh-tabVar($expr_1$) = inh-tabVar(expr)
tabVar(expr) = tabVar($expr_1$) = { }

Attribut object

object(expr) = exp

Attribut class-refer

class-refer(expr) = class-refer($expr_1$)
class-refer($expr_1$) =
Clref si ∃ ⟨$expr_1$, typevar, Clref⟩ ∈ inh-tabVar(expr)
no-value sinon

4.1.2.3. Opérateur posant une condition sur un objet

1. expr = COND(expr1, expr2)

où

$expr_1$ est un nom de classe

Attribut synt
synt(expr) =
synt($expr_1$) & synt($expr_2$)
& comp(COND,type($expr_1$), type($expr_2$))
& (object(expr1) = class)
& (object(expr2) ≠ var)

Attribut type
type(expr) = result-type(COND, type($expr_1$), type($expr_2$))

Attributs inh-ref et ref
inh-ref($expr_1$) = inh-ref(expr)
inh-ref($expr_2$) = [$expr_1$ | inh-ref(expr)]
ref(expr) = [ref($expr_2$) | ref($expr_1$)]

Attributs inh-tabVar et tabVar
inh-tabVar($expr_1$) = inh-tabVar(expr)
inh-tabVar($expr_2$) = inh-tabVar(expr)
tabVar(expr) = tabVar($expr_2$) U tabVar($expr_1$)

Attribut object
object(expr) = object($expr_1$)

Attribut class-refer
class-refer(expr) = class-refer($expr_1$)

2. expr = COND(expr1, expr2)

où
$expr_1$ est une variable

Attribut synt
synt(expr) =
synt($expr_1$) & synt($expr_2$)
& comp(COND,type($expr_1$), type($expr_2$))
& (object(expr1) = var)
& (object(expr2) ≠ var)

Attribut type
type(expr) = result-type(COND, type($expr_1$), type($expr_2$))
type($expr_1$) =
typevarInh
si ∃ ‹$expr_1$, typevarInh, ClrefInh› ∈ inh-tabVar(expr)
& si (typevar ≠ undet)
& si (∄ ‹$expr_1$, typevar, Clref› ∈ tabVar($expr_2$)
typevar
si (∄ ‹$expr_1$,typevarInh,ClrefInh› ∈ inh-tabVar(expr))
& si (∃ ‹$expr_1$, typevar, Clref› ∈ tabVar($expr_2$))
& si (typevar ≠ undet)
undet sinon

Attributs inh-ref et ref
inh-ref($expr_1$) = inh-ref(expr)
inh-ref($expr_2$) = inh-ref(expr)
ref(expr) = [ref($expr_2$) | ref($expr_1$)]

Attributs inh-tabVar et tabVar
inh-tabVar($expr_1$) = inh-tabVar(expr) U tabVar($expr_2$)
inh-tabVar($expr_2$) = inh-tabVar(expr)
tabVar(expr) = tabVar($expr_2$) U tabVar($expr_1$)

Attribut object
object(expr) = exp

Attribut class-refer
class-refer(expr) = class-refer($expr_1$)
class-refer($expr_1$) =
ClrefInh
si ∃ ‹$expr_1$, typevarInh, ClrefInh› ∈ inh-tabVar(expr)
& si (typevar ≠ undet)
& si (∄ ‹$expr_1$, typevar, Clref› ∈ tabVar($expr_2$)
Clref
si (∄ ‹$expr_1$,typevarInh,ClrefInh› ∈ inh-tabVar(expr))
& si (∃ ‹$expr_1$, typevar, Clref› ∈ tabVar($expr_2$))
& si (typevar ≠ undet)
no-value sinon

4.1.2.4. Opérateurs logiques

1. expr = NOT(expr1)

Attribut synt
synt(expr) = synt($expr_1$) & comp(NOT,type($expr_1$))

Attribut type
type(expr) = result-type(NOT, type($expr_1$))

Attributs inh-ref et ref
inh-ref($expr_1$) = inh-ref(expr)
ref(expr) = ref($expr_1$)

Attributs inh-tabVar et tabVar
inh-tabVar($expr_1$) = inh-tabVar(expr)
tabVar(expr) = tabVar($expr_1$)

Attribut object
object(expr) = cond

Attribut class-refer
class-refer(expr) = class-refer($expr_1$) = no-value

2. expr = AND(expr1, expr2)

Attribut synt
  synt(expr) =
        synt($expr_1$) & synt($expr_2$)
        & comp(AND,type($expr_1$), type($expr_2$))

Attribut type
  type(expr) = result-type(AND, type($expr_1$), type($expr_2$))

Attributs inh-ref et ref
  inh-ref($expr_1$) = inh-ref(expr)
  inh-ref($expr_2$) = [ref($expr_1$) | inh-ref(expr)]
  ref(expr) = [ref($expr_2$) | ref($expr_1$)]

Attributs inh-tabVar et tabVar
  inh-tabVar($expr_1$) = inh-tabVar(expr)
  inh-tabVar($expr_2$) = tabVar($expr_1$) U inh-tabVar(expr)
  tabVar(expr) = tabVar($expr_2$) U tabVar($expr_1$)

Attribut object
  object(expr) = cond

Attribut class-refer
  class-refer(expr) = class-refer($expr_1$) = class-refer($expr_2$) =no-value

3. expr = OR(expr1, expr2)

Attribut synt
synt(expr) =
synt($expr_1$) & synt($expr_2$)
& comp(OR,type($expr_1$), type($expr_2$))

Attribut type
type(expr) = result-type(OR, type($expr_1$), type($expr_2$))

Attributs inh-ref et ref
inh-ref($expr_1$) = inh-ref(expr)
inh-ref($expr_2$) = inh-ref(expr)
ref(expr) = [ref($expr_2$) | ref($expr_1$)]

Attributs inh-tabVar et tabVar
inh-tabVar($expr_1$) = inh-tabVar(expr)
inh-tabVar($expr_2$) = inh-tabVar(expr)
tabVar(expr) = tabVar($expr_2$) U tabVar($expr_1$)

Attribut object
object(expr) = cond

Attribut class-refer
class-refer(expr) = class-refer($expr_1$) = class-refer($expr_2$) =no-value

4. expr = EXIST-Member(expr1, expr2, expr3), FORALL-Member(expr1, expr2, expr3)

Attribut synt

synt(expr) =
synt($expr_1$) & synt($expr_2$) & synt($expr_3$)
& comp(OP,type($expr_1$),type($expr_2$),type($expr_3$))
& (object($expr_1$) = var)
& (
(object($expr_2$) ≠ var)
ou(object($expr_2$)=var& ∃ ‹$expr_2$,t2,Cl2› ∈ inh-tabVar(expr))
)
& (object($expr_3$) ≠ var)
& (∄ ‹$expr_1$, typevar, Clref› ∈ inh-tabVar(expr))

Attribut type

type(expr) = result-type(OP,type($expr_1$),type($expr_2$),type($expr_3$))
type($expr_1$) = selon la valeur de type($expr_2$) :

tuple	si type($expr_2$) = set(tuple)
classname	si type($expr_2$) = set(classname)
integer	si type($expr_2$) = set(integer)
real	si type($expr_2$) = set(real)
string	si type($expr_2$) = set(string)
undet	sinon

Attributs inh-ref et ref

inh-ref($expr_1$) = inh-ref(expr)
inh-ref($expr_2$) = inh-ref(expr)
inh-ref($expr_3$) = [ref($expr_2$) | inh-ref(expr)]
ref(expr) = [ref($expr_3$) | [ref($expr_2$) | ref($expr_1$)]]

Attributs inh-tabVar et tabVar

inh-tabVar($expr_1$) = inh-tabVar(expr)
inh-tabVar($expr_2$) = inh-tabVar(expr)
inh-tabVar($expr_3$) = inh-tabVar(expr) U { ‹expr1, type(expr1), class-refer(expr1)› }
tabVar(expr) =
tabVar($expr_3$) U tabVar($expr_2$) U tabVar($expr_1$)
U { ‹expr1, type(expr1), class-refer(expr1) › }

Attribut object

object(expr) = cond

Attribut class-refer

class-refer(expr) = no-value
class-refer($expr_1$) = class-refer($expr_2$)

5. expr = EXIST-ISIN(expr1, expr2, expr3), FORALL-ISIN(expr1, expr2, expr3)

Attribut synt

synt(expr) =

synt($expr_1$) & synt($expr_2$) & synt($expr_3$)
& comp(OP,type($expr_1$),type($expr_2$),type($expr_3$))
& (object($expr_1$) = var)
& (
(object($expr_2$) ≠ var)
ou(object($expr_2$)=var& ∃ ‹$expr_2$,t2,Cl2› ∈ inh-tabVar(expr))
)
& (object($expr_3$) ≠ var)
& (∄ ‹$expr_1$, typevar, Clref› ∈ inh-tabVar(expr))

Attribut type

type(expr) = result-type(OP,type($expr_1$),type($expr_2$),type($expr_3$))
type($expr_1$) = type($expr_2$)

Attributs inh-ref et ref

inh-ref($expr_1$) = inh-ref(expr)
inh-ref($expr_2$) = inh-ref(expr)
inh-ref($expr_3$) = [ref($expr_2$) | inh-ref(expr)]
ref(expr) = [ref($expr_3$) | [ref($expr_2$) | ref($expr_1$)]]

Attributs inh-tabVar et tabVar

inh-tabVar($expr_1$) = inh-tabVar(expr)
inh-tabVar($expr_2$) = inh-tabVar(expr)
inh-tabVar($expr_3$) = inh-tabVar(expr) U { ‹expr1, type(expr1), class-refer(expr1)› }
tabVar(expr) =
tabVar($expr_3$) U tabVar($expr_2$) U tabVar($expr_1$)
U { ‹expr1, type(expr1), class-refer(expr1) › }

Attribut object

object(expr) = cond

Attribut class-refer

class-refer(expr) = no-value
class-refer($expr_1$) = class-refer($expr_2$)

6. expr = EXIST-Included(expr1, expr2, expr3), FORALL-Included(expr1, expr2, expr3)

Attribut synt

synt(expr) =

synt($expr_1$) & synt($expr_2$) & synt($expr_3$)
& comp(OP,type($expr_1$),type($expr_2$),type($expr_3$))
& (object($expr_1$) = var)
& (
(object($expr_2$) ≠ var)
ou(object($expr_2$)=var& ∃ ‹$expr_2$,t2,Cl2› ∈ inh-tabVar(expr))
)
& (object($expr_3$) ≠ var)
& (∄ ‹$expr_1$, typevar, Clref› ∈ inh-tabVar(expr))

Attribut type

type(expr) = result-type(OP,type($expr_1$),type($expr_2$),type($expr_3$))
type($expr_1$) = type($expr_2$)

Attributs inh-ref et ref

inh-ref($expr_1$) = inh-ref(expr)
inh-ref($expr_2$) = inh-ref(expr)
inh-ref($expr_3$) = [ref($expr_2$) | inh-ref(expr)]
ref(expr) = [ref($expr_3$) | [ref($expr_2$) | ref($expr_1$)]]

Attributs inh-tabVar et tabVar

inh-tabVar($expr_1$) = inh-tabVar(expr)
inh-tabVar($expr_2$) = inh-tabVar(expr)
inh-tabVar($expr_3$) = inh-tabVar(expr) U { ‹expr1, type(expr1), class-refer(expr1)› }
tabVar(expr) =
tabVar($expr_3$) U tabVar($expr_2$) U tabVar($expr_1$)
U { ‹expr1, type(expr1), class-refer(expr1) › }

Attribut object

object(expr) = cond

Attribut class-refer

class-refer(expr) = no-value
class-refer($expr_1$) = class-refer($expr_2$)

4.1.2.5. Opérateurs de comparaison

1. expr = Cop(expr1, expr2)

où

- Cop est l'un des opérateurs : EQ, NE, GT, GE, ST, SE
- $expr_1$ et $expr_2$ ne sont pas des variables.

Attribut synt
synt(expr) =
synt($expr_1$) & synt($expr_2$)
& comp(Cop,type($expr_1$), type($expr_2$))
& (object($expr_1$) ≠ var)
& (object($expr_2$) ≠ var)

Attribut type
type(expr) = result-type(Cop, type($expr_1$), type($expr_2$))

Attributs inh-ref et ref
inh-ref($expr_1$) = inh-ref(expr)
inh-ref($expr_2$) = [ref($expr_1$) | inh-ref(expr)]
ref(expr) = [ref($expr_2$) | ref($expr_1$)]

Attributs inh-tabVar et tabVar
inh-tabVar($expr_1$) = inh-tabVar(expr)
inh-tabVar($expr_2$) = tabVar($expr_1$) U inh-tabVar(expr)
tabVar(expr) = tabVar($expr_2$) U tabVar($expr_1$)

Attribut object
object(expr) = cond

Attribut class-refer
class-refer(expr) = no-value

2. expr = Cop(expr1, expr2)

où

- Cop est l'un des opérateurs : NE, GT, GE, ST, SE
- $expr_1$ est une variable, $expr_2$ n'est pas une variable

Attribut synt
synt(expr) =
synt($expr_1$) & synt($expr_2$)
& comp(Cop,type($expr_1$), type($expr_2$))
& (object($expr_1$) = var)
& (object($expr_2$) ≠ var)
& (∃ ‹$expr_1$, typevar, Clref› ∈ inh-tabVar(expr))

Attribut type
type(expr) = result-type(Cop, type($expr_1$), type($expr_2$))
type($expr_1$) =

typevar	si ∃ ‹$expr_1$, typevar, Clref› ∈ inh-tabVar(expr)
undet	sinon

Attributs inh-ref et ref
inh-ref($expr_1$) = inh-ref(expr)
inh-ref($expr_2$) = [ref($expr_1$) | inh-ref(expr)]
ref(expr) = [ref($expr_2$) | ref($expr_1$)]

Attributs inh-tabVar et tabVar
inh-tabVar($expr_1$) = inh-tabVar(expr)
inh-tabVar($expr_2$) = inh-tabVar(expr)
tabVar(expr) = tabVar($expr_2$) U tabVar($expr_1$)

Attribut object
object(expr) = cond

Attribut class-refer
class-refer(expr) = no-value
class-refer($expr_1$) =

Clref	si ∃ ‹$expr_1$, typevar, Clref› ∈ inh-tabVar(expr)
no-value	sinon

3. expr = Cop(expr1, expr2)

où

- Cop est l'un des opérateurs : NE, GT, GE, ST, SE
- $expr_1$ est une variable, $expr_2$ est une variable

Attribut synt
synt(expr) =
synt($expr_1$) & synt($expr_2$)
& comp(Cop,type($expr_1$), type($expr_2$))
& (object($expr_1$) = var)
& (object($expr_2$) = var)
& (∃ ‹$expr_1$, type1,Clref1› ∈ inh-tabVar(expr)
& (∃ ‹$expr_2$, type2, Clref2› ∈ inh-tabVar(expr))

Attribut type
type(expr) = result-type(Cop, type($expr_1$), type($expr_2$))
type($expr_1$) =
type1 si ∃ ‹$expr_1$, type1, Clref1› ∈ inh-tabVar(expr)
undet sinon
type($expr_2$) =
type2 si ∃ ‹$expr_2$, type2, Clref2› ∈ inh-tabVar(expr)
undet sinon

Attributs inh-ref et ref
inh-ref($expr_1$) = inh-ref(expr)
inh-ref($expr_2$) = [ref($expr_1$) | inh-ref(expr)]
ref(expr) = [ref($expr_2$) | ref($expr_1$)]

Attributs inh-tabVar et tabVar
inh-tabVar($expr_1$) = inh-tabVar(expr)
inh-tabVar($expr_2$) = inh-tabVar(expr)
tabVar(expr) = tabVar($expr_2$) U tabVar($expr_1$)

Attribut object
object(expr) = cond

Attribut class-refer
class-refer(expr) = no-value
class-refer($expr_1$) =
Clref1 si ∃ ‹$expr_1$, type1, Clref1› ∈ inh-tabVar(expr)
no-value sinon
class-refer($expr_2$) =
Clref2 si ∃ ‹$expr_2$, type2, Clref2› ∈ inh-tabVar(expr)
no-value sinon

4. expr = Cop(expr1, expr2)

où

- Cop est l'un des operateurs : EQ, NE, GT, GE, ST, SE
- $expr_1$ n'est pas une variable, $expr_2$ est une variable

Attribut synt
synt(expr) =
synt($expr_1$) & synt($expr_2$)
& comp(Cop,type($expr_1$), type($expr_2$))
& (object($expr_1$) ≠ var)
& (object($expr_2$) = var)
& (∃ ‹$expr_2$, type2, Clref2› ∈ inh-tabVar(expr))

Attribut type
type(expr) = result-type(Cop, type($expr_1$), type($expr_2$))
type($expr_2$) =
type2 si ∃ ‹$expr_2$, type2, Clref2› ∈ inh-tabVar(expr)
undet sinon

Attributs inh-ref et ref
inh-ref($expr_1$) = inh-ref(expr)
inh-ref($expr_2$) = [ref($expr_1$) | inh-ref(expr)]
ref(expr) = [ref($expr_2$) | ref($expr_1$)]

Attributs inh-tabVar et tabVar
inh-tabVar($expr_1$) = inh-tabVar(expr)
inh-tabVar($expr_2$) = inh-tabVar(expr)
tabVar(expr) = tabVar($expr_2$) U tabVar($expr_1$)

Attribut object
object(expr) = cond

Attribut class-refer
class-refer(expr) = no-value
class-refer($expr_2$) =
Clref2 si ∃ ‹$expr_2$, type2, Clref2› ∈ inh-tabVar(expr)
no-value sinon

5. expr = Cop(expr1, expr2)

où

- Cop est l'opérateur : EQ
- $expr_1$ est une variable, $expr_2$ n'est pas une variable

Attribut synt
synt(expr) =
synt($expr_1$) & synt($expr_2$)
& comp(Cop,type($expr_1$), type($expr_2$))
& (object($expr_1$) = var)
& (object($expr_2$) ≠ var)

Attribut type
type(expr) = result-type(Cop, type($expr_1$), type($expr_2$))
type($expr_1$) =
typevar si ∃ ‹$expr_1$, typevar, Clref› ∈ inh-tabVar(expr)
type($expr_2$) si ∄ ‹$expr_1$, typevar, Clref› ∈ inh-tabVar(expr), & type($expr_2$) ≠ undet
undet sinon

Attributs inh-ref et ref
inh-ref($expr_1$) = inh-ref(expr)
inh-ref($expr_2$) = [ref($expr_1$) | inh-ref(expr)]
ref(expr) = [ref($expr_2$) | ref($expr_1$)]

Attributs inh-tabVar et tabVar
inh-tabVar($expr_1$) = inh-tabVar(expr)
inh-tabVar($expr_2$) = inh-tabVar(expr)
tabVar(expr) = tabVar($expr_2$) U tabVar($expr_1$)

Attribut object
object(expr) = cond

Attribut class-refer
class-refer(expr) = no-value
class-refer($expr_1$) =
Clref si ∃ ‹$expr_1$, typevar, Clref› ∈ inh-tabVar(expr)
class-refer($expr_2$) si ∄ ‹$expr_1$, typevar, Clref› ∈ inh-tabVar(expr) & class-refer($expr_2$) ≠ no-value
no-value sinon

6. expr = Cop(expr1, expr2)

où

- Cop est l'opérateur : EQ,
- $expr_1$ est une variable, $expr_2$ est une variable

Attribut synt

synt(expr) =

synt($expr_1$) & synt($expr_2$)
& comp(Cop,type($expr_1$), type($expr_2$))
& (object($expr_1$) = var)
& (object($expr_2$) = var)
& (∃ ‹$expr_2$, type2, Clref2› ∈ inh-tabVar(expr))

Attribut type

type(expr) = result-type(Cop, type($expr_1$), type($expr_2$))

type($expr_1$) =

typeI si ∃ ‹$expr_1$, typeI, ClrefI› ∈ inh-tabVar(expr)
type($expr_2$) si ∄ ‹$expr_1$, typevar, Clref› ∈ inh-tabVar(expr), & type($expr_2$) ≠ undet
undet sinon

type($expr_2$) =

type2 si ∃ ‹$expr_2$, type2, Clref2› ∈ inh-tabVar(expr)
undet sinon

Attributs inh-ref et ref

inh-ref($expr_1$) = inh-ref(expr)
inh-ref($expr_2$) = [ref($expr_1$) | inh-ref(expr)]
ref(expr) = [ref($expr_2$) | ref($expr_1$)]

Attributs inh-tabVar et tabVar

inh-tabVar($expr_1$) = inh-tabVar(expr)
inh-tabVar($expr_2$) = inh-tabVar(expr)
tabVar(expr) = tabVar($expr_2$) U tabVar($expr_1$)

Attribut object

object(expr) = cond

Attribut class-refer

class-refer(expr) = no-value

class-refer($expr_1$) =

ClrefI si ∃ ‹$expr_1$, typeI, ClrefI› ∈ inh-tabVar(expr)
class-refer($expr_2$) si ∄ ‹$expr_1$, typevar, Clref› ∈ inh-tabVar(expr) & class-refer($expr_2$) ≠ no-value
no-value sinon

class-refer($expr_2$) =

Clref2 si ∃ ‹$expr_2$, type2, Clref2› ∈ inh-tabVar(expr)
no-value sinon

4.1.2.6. Opérateurs ensemblistes

1. expr = setOp(expr1, expr2)

où

- setOp est l'opérateur : Member, Included, EqSet
- $expr_1$ n'est pas une variable, expr2 n'est pas une variable

Attribut synt
synt(expr) =
synt($expr_1$) & synt($expr_2$)
& comp(setOp,type($expr_1$), type($expr_2$))
& (object($expr_1$) ≠ var)
& (object($expr_2$) ≠ var)

Attribut type
type(expr) = result-type(setOp, type($expr_1$), type($expr_2$))

Attributs inh-ref et ref
inh-ref($expr_1$) = inh-ref(expr)
inh-ref($expr_2$) = [ref($expr_1$) | inh-ref(expr)]
ref(expr) = [ref($expr_2$) | ref($expr_1$)]

Attributs inh-tabVar et tabVar
inh-tabVar($expr_1$) = inh-tabVar(expr)
inh-tabVar($expr_2$) = tabVar($expr_1$) U inh-tabVar(expr)
tabVar(expr) = tabVar($expr_2$) U tabVar($expr_1$)

Attribut object
object(expr) = cond

Attribut class-refer
class-refer(expr) = no-value

2. expr = setOp(expr1, expr2)

où

- setOp est l'opérateur : ISIN
- $expr_1$ n'est pas une variable, expr2 n'est pas une variable

Attribut synt

synt(expr) =
synt($expr_1$) & synt($expr_2$)
& comp(setOp,type($expr_1$), type($expr_2$))
& (object($expr_1$) ≠ var)
& (object($expr_2$) ≠ var)
& (expr2 = SET(expr3)
ou expr2 = COND(classname, expr4)
ou expr2 = classname
)

Attribut type

type(expr) = result-type(setOp, type($expr_1$), type($expr_2$))

Attributs inh-ref et ref

inh-ref($expr_1$) = inh-ref(expr)
inh-ref($expr_2$) = [ref($expr_1$) | inh-ref(expr)]
ref(expr) = [ref($expr_2$) | ref($expr_1$)]

Attributs inh-tabVar et tabVar

inh-tabVar($expr_1$) = inh-tabVar(expr)
inh-tabVar($expr_2$) = tabVar($expr_1$) U inh-tabVar(expr)
tabVar(expr) = tabVar($expr_2$) U tabVar($expr_1$)

Attribut object

object(expr) = cond

Attribut class-refer

class-refer(expr) = no-value

3. expr = setOp(expr1, expr2)

où

- setOp est l'opérateur : ISIN
- $expr_1$ est une variable, $expr_2$ n'est pas une variable

Attribut synt

synt(expr) =
synt($expr_1$) & synt($expr_2$)
& comp(setOp,type($expr_1$), type($expr_2$))
& (object($expr_1$) = var)
& (object($expr_2$) ≠ var)
& (∄ ‹$expr_1$, typevar, Clref› ∈ inh-tabVar(expr))
& (expr2 = SET(expr3)
ou expr2 = COND(classname, expr4)
ou expr2 = classname
)

Attribut type

type(expr) = result-type(setOp, type($expr_1$), type($expr_2$))
type($expr_1$) = type($expr_2$)

Attributs inh-ref et ref

inh-ref($expr_1$) = inh-ref(expr)
inh-ref($expr_2$) = inh-ref(expr)
ref(expr) = [ref($expr_2$) | ref($expr_1$)]

Attributs inh-tabVar et tabVar

inh-tabVar($expr_1$) = inh-tabVar(expr)
inh-tabVar($expr_2$) = inh-tabVar(expr)
tabVar(expr) = { ‹$expr_1$, type($expr_1$), class-refer($expr_1$) › }

Attribut object

object(expr) = cond

Attribut class-refer

class-refer(expr) = no-value
class-refer($expr_1$) = class-refer($expr_2$)

4. expr = setOp(expr1, expr2)

où

- setOp est l'opérateur : ISIN
- $expr_1$ n'est pas une variable, expr2 est une variable

Attribut synt

synt(expr) =
synt($expr_1$) & synt($expr_2$)
& comp(setOp,type($expr_1$), type($expr_2$))
& (object($expr_1$) ≠ var)
& (object($expr_2$) = var)
& ∃ ‹$expr_2$, type2, Clref2› ∈ inh-tabVar(expr))

Attribut type

type(expr) = result-type(setOp, type($expr_1$), type($expr_2$))

Attributs inh-ref et ref

inh-ref($expr_1$) = inh-ref(expr)
inh-ref($expr_2$) = [ref($expr_1$) | inh-ref(expr)]
ref(expr) = [ref($expr_2$) | ref($expr_1$)]

Attributs inh-tabVar et tabVar

inh-tabVar($expr_1$) = inh-tabVar(expr)
inh-tabVar($expr_2$) = tabVar($expr_1$) U inh-tabVar(expr)
tabVar(expr) = tabVar($expr_2$) U tabVar($expr_1$)

Attribut object

object(expr) = cond

Attribut class-refer

class-refer(expr) = no-value

5. expr = setOp(expr1, expr2)

où

- setOp est l'opérateur : ISIN
- $expr_1$ est une variable, $expr_2$ est une variable

Attribut synt

synt(expr) =
synt($expr_1$) & synt($expr_2$)
& comp(setOp,type($expr_1$), type($expr_2$))
& (object($expr_1$) = var)
& (object($expr_2$) ≠ var)
& (∄ ‹$expr_1$, typevar, Clref› ∈ inh-tabVar(expr))
& (object($expr_2$) = var)
& ∃ ‹$expr_2$, type2, Clref2› ∈ inh-tabVar(expr))

Attribut type

type(expr) = result-type(setOp, type($expr_1$), type($expr_2$))
type($expr_1$) = type($expr_2$)

Attributs inh-ref et ref

inh-ref($expr_1$) = inh-ref(expr)
inh-ref($expr_2$) = inh-ref(expr)
ref(expr) = [ref($expr_2$) | ref($expr_1$)]

Attributs inh-tabVar et tabVar

inh-tabVar($expr_1$) = inh-tabVar(expr)
inh-tabVar($expr_2$) = inh-tabVar(expr)
tabVar(expr) = { ‹$expr_1$, type($expr_1$), class-refer($expr_1$) › }

Attribut object

object(expr) = cond

Attribut class-refer

class-refer(expr) = no-value
class-refer($expr_1$) = class-refer($expr_2$)

6. expr = setOp(expr1, expr2)

où

- setOp est l'opérateur : Member, Included
- $expr_1$ est une variable, $expr_2$ n'est pas une variable

Attribut synt

synt(expr) =

synt($expr_1$) & synt($expr_2$)
& comp(setOp,type($expr_1$), type($expr_2$))
& (object($expr_1$) = var)
& (object($expr_2$) ≠ var)
& (∃ ⟨$expr_1$, typel, $Clref_1$⟩ ∈ inh-tabVar(expr))

Attribut type

type(expr) = result-type(setOp, type($expr_1$), type($expr_2$))

type($expr_1$) =

typel	si ∃ ⟨$expr_1$, typel, $Clref_1$⟩ ∈ inh-tabVar(expr)
undet	sinon

Attributs inh-ref et ref

inh-ref($expr_1$) = inh-ref(expr)
inh-ref($expr_2$) = inh-ref(expr)
ref(expr) = [ref($expr_2$) | ref($expr_1$)]

Attributs inh-tabVar et tabVar

inh-tabVar($expr_1$) = inh-tabVar(expr)
inh-tabVar($expr_2$) = inh-tabVar(expr)
tabVar(expr) = tabVar($expr_2$) U tabVar($expr_1$)

Attribut object

object(expr) = cond

Attribut class-refer

class-refer(expr) = no-value

class-refer($expr_1$) =

Clref1	si ∃ ⟨$expr_1$, typel, $Clref_1$⟩ ∈ inh-tabVar(expr)
no-value	sinon

7. expr = setOp(expr1, expr2)

où
- setOp est l'opérateur : EqSet
- $expr_1$ est une variable, $expr_2$ n'est pas une variable

Attribut synt
synt(expr) =
synt($expr_1$) & synt($expr_2$)
& comp(setOp,type($expr_1$), type($expr_2$))
& (object($expr_1$) = var) & (object($expr_2$) ≠ var)

Attribut type
type(expr) = result-type(setOp, type($expr_1$), type($expr_2$))
type($expr_1$) = selon la valeur de inh-tabVar(expr) :
- typevar
  si ∃ ‹$expr_1$, typevar, ClrefInh› ∈ inh-tabVar(expr) & (∄ ‹$expr_1$, typel, Clref› ∈ tabVar($expr_2$)
- type($expr_2$)
  si ∄ ‹$expr_1$, typevar, ClrefInh› ∈ inh-tabVar(expr)
- undet sinon

Attributs inh-ref et ref
inh-ref($expr_1$) = inh-ref(expr)
inh-ref($expr_2$) = inh-ref(expr)
ref(expr) = [ref($expr_2$) | ref($expr_1$)]

Attributs inh-tabVar et tabVar
inh-tabVar($expr_1$) = inh-tabVar(expr)
inh-tabVar($expr_2$) = inh-tabVar(expr)
tabVar(expr) = selon la valeur de inh-tabVar(expr) :
- tabVar($expr_2$) U tabVar($expr_1$)
  si ∃ ‹$expr_1$, typevar, ClrefInh› ∈ inh-tabVar(expr) & (∄ ‹$expr_1$, typel, Clref› ∈ tabVar($expr_2$)
- tabVar($expr_2$) U tabVar($expr_1$) U { ‹$expr_1$, type($expr_1$), class-refer(expr1)› }
  si ∄ ‹$expr_1$, typevar, ClrefInh› ∈ inh-tabVar(expr)

Attribut object
object(expr) = cond

Attribut class-refer
class-refer(expr) = no-value
class-refer($expr_1$) = selon la valeur de inh-tabVar(expr) :
- ClrefInh
  si ∃ ‹$expr_1$, typevar, ClrefInh› ∈ inh-tabVar(expr) & (∄ ‹$expr_1$, typel, Clref› ∈ tabVar($expr_2$)
- class-refer($expr_2$)
  si ∄ ‹$expr_1$, typevar, ClrefInh› ∈ inh-tabVar(expr)
- no-value sinon

8. expr = setOp(expr1, expr2)

où

- setOp est l'opérateur : Member, Included, EqSet
- $expr_1$ n'est pas une variable, expr2 est une variable

Attribut synt
synt(expr) =
synt($expr_1$) & synt($expr_2$)
& comp(setOp,type($expr_1$), type($expr_2$))
& (object($expr_1$) ≠ var)
& (object($expr_2$) = var)
& (∃ ‹$expr_2$, type2, $Clref_2$› ∈ inh-tabVar(expr)

Attribut type
type(expr) = result-type(setOp, type($expr_1$), type($expr_2$))
type($expr_2$) =
type2 si ∃ ‹$expr_2$, type2, $Clref_2$› ∈ inh-tabVar(expr)
undet sinon

Attributs inh-ref et ref
inh-ref($expr_1$) = inh-ref(expr)
inh-ref($expr_2$) = inh-ref(expr)
ref(expr) = [ref($expr_2$) | ref($expr_1$)]

Attributs inh-tabVar et tabVar
inh-tabVar($expr_1$) = inh-tabVar(expr)
inh-tabVar($expr_2$) = tabVar($expr_1$) U inh-tabVar(expr)
tabVar(expr) = tabVar($expr_2$) U tabVar($expr_1$)

Attribut object
object(expr) = cond

Attribut class-refer
class-refer(expr) = no-value
class-refer($expr_2$) =
Clref2 si ∃ ‹$expr_2$, type2, $Clref_2$› ∈ inh-tabVar(expr)
no-value sinon

9. expr = setOp(expr1, expr2)

où

- setOp est l'opérateur : Member, Included
- $expr_1$ est une variable, expr2 est une variable

Attribut synt

synt(expr) =
synt($expr_1$) & synt($expr_2$)
& comp(setOp,type($expr_1$), type($expr_2$))
& (object($expr_1$) = var)
& (object($expr_2$) = var)
& (∃ ‹$expr_1$, type1, Clref$_1$› ∈ inh-tabVar(expr)
& (∃ ‹$expr_2$, type2, Clref$_2$› ∈ inh-tabVar(expr)

Attribut type

type(expr) = result-type(setOp, type($expr_1$), type($expr_2$))
type($expr_1$) =

type1	si ∃ ‹$expr_1$, type1, Clref$_1$› ∈ inh-tabVar(expr)
undet	sinon

type($expr_2$) =

type2	si ∃ ‹$expr_2$, type2, Clref$_2$› ∈ inh-tabVar(expr)
undet	sinon

Attributs inh-ref et ref :

inh-ref($expr_1$) = inh-ref(expr)
inh-ref($expr_2$) = inh-ref(expr)
ref(expr) = [ref($expr_2$) | ref($expr_1$)]

Attributs inh-tabVar et tabVar

inh-tabVar($expr_1$) = inh-tabVar(expr)
inh-tabVar($expr_2$) = tabVar($expr_1$) U inh-tabVar(expr)
tabVar(expr) = tabVar($expr_2$) U tabVar($expr_1$)

Attribut object

object(expr) = cond

Attribut class-refer

class-refer(expr) = no-value
class-refer($expr_1$) =

Clref1	si ∃ ‹$expr_1$, type1, Clref$_1$› ∈ inh-tabVar(expr)
no-value	sinon

class-refer($expr_2$) =

Clref2	si ∃ ‹$expr_2$, type2, Clref$_2$› ∈ inh-tabVar(expr)
no-value	sinon

10. expr = setOp(expr1, expr2)

où

- setOp est l'opérateur : EqSet
- $expr_1$ est une variable, $expr_2$ est une variable

Attribut synt

synt(expr) =
$synt(expr_1)$ & $synt(expr_2)$
& comp(setOp,$type(expr_1)$, $type(expr_2)$)
& ($object(expr_1)$ = var)
& ($object(expr_2)$ = var)
& (∃ ‹$expr_2$, type2, $Clref_2$› ∈ inh-tabVar(expr)

Attribut type

type(expr) = result-type(setOp, $type(expr_1)$, $type(expr_2)$)
$type(expr_1)$ = selon la valeur de inh-tabVar(expr) :
- typevar
  si ∃ ‹$expr_1$, typevar, ClrefInh› ∈ inh-tabVar(expr) & (∄ ‹$expr_1$, type1, Clref› ∈ $tabVar(expr_2)$
- $type(expr_2)$
  si ∄ ‹$expr_1$, typevar, ClrefInh› ∈ inh-tabVar(expr)
- undet sinon

$type(expr_2)$ =
type2 si ∃ ‹$expr_2$, type2, $Clref_2$› ∈ inh-tabVar(expr)
undet sinon

Attributs inh-ref et ref

$inh\text{-}ref(expr_1)$ = inh-ref(expr)
$inh\text{-}ref(expr_2)$ = inh-ref(expr)
ref(expr) = [$ref(expr_2)$ | $ref(expr_1)$]

Attributs inh-tabVar et tabVar

$inh\text{-}tabVar(expr_1)$ = inh-tabVar(expr)
$inh\text{-}tabVar(expr_2)$ = inh-tabVar(expr)
tabVar(expr) = selon la valeur de inh-tabVar(expr) :
- $tabVar(expr_2)$ U $tabVar(expr_1)$
  si ∃ ‹$expr_1$, typevar, ClrefInh› ∈ inh-tabVar(expr) & (∄ ‹$expr_1$, type1, Clref› ∈ $tabVar(expr_2)$
- $tabVar(expr_2)$ U $tabVar(expr_1)$ U { ‹$expr_1$, $type(expr_1)$, class-refer(expr1)› }
  si ∄ ‹$expr_1$, typevar, ClrefInh› ∈ inh-tabVar(expr)

Attribut object

object(expr) = cond

Attribut class-refer

class-refer(expr) = no-value

class-refer($expr_1$) = selon la valeur de inh-tabVar(expr) :

- ClrefInh
  si $\exists$ ‹$expr_1$, typevar, ClrefInh› $\in$ inh-tabVar(expr) & ($\nexists$ ‹$expr_1$, type1, Clref› $\in$ tabVar($expr_2$)
- class-refer($expr_2$)
  si $\nexists$ ‹$expr_1$, typevar, ClrefInh› $\in$ inh-tabVar(expr)
- no-value sinon

class-refer($expr_2$) =

Clref2 si $\exists$ ‹$expr_2$, type2, $Clref_2$› $\in$ inh-tabVar(expr)

no-value sinon

4.1.2.7. Opérateurs définissant des fonctions agrégées

1. expr = FUNCT(expr1)

où

- $expr_1$ n'est pas une variable
- FUNCT est l'un des opérateurs : AVG, MIN, MAX, SUM, COUNT

Attribut synt
synt(expr) =
synt($expr_1$)
& comp(FUNCT,type($expr_1$))
& object(expr1) ≠ var

Attribut type
type(expr) = result-type(FUNCT, type($expr_1$))

Attributs inh-ref et ref
inh-ref($expr_1$) = inh-ref(expr)
ref(expr) = ref($expr_1$)

Attributs inh-tabVar et tabVar
inh-tabVar($expr_1$) = inh-tabVar(expr)
tabVar(expr) = tabVar($expr_1$)

Attribut object
object(expr) = exp

Attribut class-refer
class-refer(expr) = no-value

2. expr = FUNCT(expr1)

où

$expr_1$ est une variable

Attribut synt

synt(expr) =

synt($expr_1$)
& comp(FUNCT,type($expr_1$))
& object(expr1) = var
& (∃ ⟨$expr_1$, type1, Clref1⟩ ∈ inh-tabVar(expr)

Attribut type

type(expr) = result-type(FUNCT, type($expr_1$))

type($expr_1$) =

type1	si ∃ ⟨$expr_1$, type1, Clref1⟩ ∈ inh-tabVar(expr)
undet	sinon

Attributs inh-ref et ref

inh-ref($expr_1$) = inh-ref(expr)
ref(expr) = ref($expr_1$)

Attributs inh-tabVar et tabVar

inh-tabVar($expr_1$) = inh-tabVar(expr)
tabVar(expr) = tabVar($expr_1$)

Attribut object

object(expr) = exp

Attribut class-refer

class-refer(expr) = no-value

class-refer($expr_1$) =

Clref1	si ∃ ⟨$expr_1$, type1, $Clref_1$⟩ ∈ inh-tabVar(expr)
no-value	sinon

4.1.2.8. Opérateurs arithmétiques

1. expr = AOp(expr1, expr2)

où

- AOp est l'un des opérateurs : PLUS, MINUS, TIMES, DIV
- $expr_1$, $expr_2$ ne sont pas des variables

Attribut synt
synt(expr) =
synt($expr_1$) & synt($expr_2$)
& comp(AOp,type($expr_1$), type($expr_2$))
& (object($expr_1$) ≠ var)
& (object($expr_2$) ≠ var)

Attribut type
type(expr) = result-type(AOp, type($expr_1$), type($expr_2$))

Attributs inh-ref et ref
inh-ref($expr_1$) = inh-ref(expr)
inh-ref($expr_2$) = [ref($expr_1$) | inh-ref(expr)]
ref(expr) = [ref($expr_2$) | ref($expr_1$)]

Attributs inh-tabVar et tabVar
inh-tabVar($expr_1$) = inh-tabVar(expr)
inh-tabVar($expr_2$) = tabVar($expr_1$) U inh-tabVar(expr)
tabVar(expr) = tabVar($expr_2$) U tabVar($expr_1$)

Attribut object
object(expr) = exp

Attribut class-refer
class-refer(expr) = no-value

2. expr = AOp(expr1, expr2)

où

- AOp est l'un des opérateurs : PLUS, MINUS, TIMES, DIV
- $expr_1$ est une variable, $expr_2$ n'est pas une variable

Note :

Le cas où $expr_2$ est une variable, $expr_1$ n'est pas une variable est similaire à celui-ci.

Attribut synt

synt(expr) =

synt($expr_1$) & synt($expr_2$)
& comp(AOp,type($expr_1$), type($expr_2$))
& (object($expr_1$) = var)
& (object($expr_2$) ≠ var)
& (∃ ‹$expr_1$, typevar, Clref1› ∈ inh-tabVar(expr))

Attribut type

type(expr) = result-type(AOp, type($expr_1$), type($expr_2$))

type($expr_1$) =

typevar	si ∃ ‹$expr_1$, typevar, Clref1› ∈ inh-tabVar(expr)
undet	sinon

Attributs inh-ref et ref

inh-ref($expr_1$) = inh-ref(expr)
inh-ref($expr_2$) = [ref($expr_1$) | inh-ref(expr)]
ref(expr) = [ref($expr_2$) | ref($expr_1$)]

Attributs inh-tabVar et tabVar

inh-tabVar($expr_1$) = inh-tabVar(expr)
inh-tabVar($expr_2$) = tabVar($expr_1$) U inh-tabVar(expr)
tabVar(expr) = tabVar($expr_2$) U tabVar($expr_1$)

Attribut object

object(expr) = exp

Attribut class-refer

class-refer(expr) = no-value

class-refer($expr_1$) =

Clref1	si ∃ ‹$expr_1$, typevar, $Clref_1$› ∈ inh-tabVar(expr)
no-value	sinon

3. expr = AOp(expr1, expr2)

où

- AOp est l'un des opérateurs : PLUS, MINUS, TIMES, DIV
- $expr_1$ est une variable, $expr_2$ est une variable

Attribut synt
  synt(expr) =
        synt($expr_1$) & synt($expr_2$)
        & comp(AOp,type($expr_1$), type($expr_2$))
        & (object($expr_1$) = var)
        & (object($expr_2$) = var)
        & (∃ ‹$expr_1$, type1, Clref1› ∈ inh-tabVar(expr)
        & (∃ ‹$expr_2$, type2, Clref2› ∈ inh-tabVar(expr))

Attribut type
  type(expr) = result-type(Cop, type($expr_1$), type($expr_2$))
  type($expr_1$) =
        type1 si ∃ ‹$expr_1$, type1, Clref1› ∈ inh-tabVar(expr)
        undet sinon
  type($expr_2$) =
        type2 si ∃ ‹$expr_2$, type2, Clref2› ∈ inh-tabVar(expr)
        undet sinon

Attributs inh-ref et ref
  inh-ref($expr_1$) = inh-ref(expr)
  inh-ref($expr_2$) = inh-ref(expr)
  ref(expr) = [ref($expr_2$) | ref($expr_1$)]

Attributs inh-tabVar et tabVar
  inh-tabVar($expr_1$) = inh-tabVar(expr)
  inh-tabVar($expr_2$) = inh-tabVar(expr)
  tabVar(expr) = tabVar($expr_2$) U tabVar($expr_1$)

Attribut object
  object(expr) = cond

Attribut class-refer
  class-refer(expr) = no-value
  class-refer($expr_1$) =
        Clref1 si ∃ ‹$expr_1$, type1, $Clref_1$› ∈ inh-tabVar(expr)
        no-value sinon
  class-refer($expr_2$) =
        Clref1 si ∃ ‹$expr_2$, type2, $Clref_2$› ∈ inh-tabVar(expr)
        no-value sinon

4.2. Correction syntaxique d'un slot

Soit une définition de classe Cl contenant un slot S.

Le slot S slot possède une facette (def, PCLexpression), pour laquelle on désire vérifier la correction syntaxique.

expr = PCLexpression

Attribut synt
synt(expr) = synt(PCLexpression)

Attribut type
type(expr) = type(PCLexpression)

Attributs inh-ref et ref
inh-ref(PCLexpression) = [Cl]
ref(expr) = ref(PCLexpression)

Attributs inh-tabVar et tabVar
inh-tabVar(PCLexpression) = { }
tabVar(expr) = tabVar(PCLexpression)

Attribut object
object(expr) =
- exp pour un slot de propriété
- cond pour un slot de contrainte

Attribut class-refer
class-refer(expr) = class-refer(PCLexpression)

4.3. Correction syntaxique d'une requête

Soit une requête, pour laquelle on désire vérifier la correction syntaxique.

expr = PCLquery

Attribut synt
synt(expr) = synt(PCLquery)

Attribut type
type(expr) = type(PCLquery)

Attributs inh-ref et ref
inh-ref(PCLquery) = [ ]
ref(expr) = ref(PCLquery)

Attributs inh-tabVar et tabVar
inh-tabVar(PCLquery) = { }
tabVar(expr) = tabVar(PCLquery)

Attribut object
object(expr) = exp

Attribut class-refer
class-refer(expr) = class-refer(PCLquery)

Relations de dépendance

Annexe 6

1. Notion de relations de dépendance

L'annexe 3 ("Contexte de définition des expressions") insiste sur l'importance d'une liste de noms de classes pouvant être référencés par une expression.

De plus, le langage PROBE, pour pouvoir établir certaines vérifications syntaxiques dont il a la charge, a besoin de disposer d'une liste plus explicite de ces noms de classes. Ainsi, chaque nom de slot doit être relevé, en précisant la classe dans laquelle il est défini.

L'ensemble des relations de dépendance d'une expression peut être décrit par tous les noms de classes apparaissant explicitement dans l'expression, et tous les noms de classes implicitement référencés par une sous-expression (généralement un nom de slot, une variable).

A cet effet, on désignera l'ensemble des relations de dépendance d'une expression par une liste d'éléments qui sont soit un nom de classe, soit un couple de la forme (nom de slot, nom de classe).

Chaque nom de classe apparaissant dans cette liste peut être :

- un nom de classe apparaissant explicitement dans l'expression,
- un nom de classe référencé par une sous-expression (résultat de l'attribut class-refer de la grammaire attribuée définissant la correction syntaxique des expressions.

La liste de relations de dépendance peut être définie, pour chaque expression, au moyen d'un attribut, class-depend, ajouté à la grammaire attribuée pour la correction syntaxique des expressions.

Cet attribut est défini de la manière suivante :

Attribut "class-depend"

class-depend : Expr ----------> ListofClass + { undef[ined] }

exp ~~~~~~~~> class-depend(exp)

Où les différents ensembles sont :

- Expr = { expressions }
- ListofClass = { [x | y] tel que x est soit un nom de classe, soit un couple (slot, class contenant le slot), et y un élément de ListofClass }

La fonction **class-depend** associe à chaque expression exp une valeur class-depend(exp) qui est :

class-depend(exp) =

la liste des relations de dépendance de l'expression exp, au sens défini ci-dessus :

une liste d'éléments qui sont soit un nom de classe, soit un couple (slot, classe contenant le slot).

undef[ined] si l'expression n'est pas syntaxiquement correcte.

2. Définition des relations de dépendance

Remarquons que cet attribut étant défini comme un attribut supplémentaire de la grammaire attribuée pour la correction syntaxique des expressions, on ne tient compte ici que de la valuer de l'attribut pour des expressions syntaxiquement correctes.

Dans tous les cas où une erreur de syntaxe est détectée, l'attribut class-depend prend la valeur undef[ined].

2.1. Relations de dépendance d'une expression

2.1.1. Expression de base

1. expr = constante

Attribut class-depend
class-depend(expr) = [ ]

2. expr = classname

Attribut class-depend
class-depend(expr) = [classname]

3. expr = variable

Attribut class-depend
class-depend(variable) =
[clref]
si ∃ ‹variable, typevar, Clref› ∈ inh-tabVar(expr)

undefined
sinon

4. expr = slotname

Attribut class-depend
class-depend(expr) =
[(slotname, Clref]
si ∃ t_i ∈ inh-ref(expr) où t_i est le premier tel que :
(slotname ∈ classe t_i)

undefined
sinon

5. expr = (t1, ..., tn)

Attribut class-depend

class-depend(expr) = class-depend(t_1) U ..... U class-depend(t_n)

6. expr = [c1, ..., cn]

Attribut class-depend

class-depend(expr) = [ ]

2.1.2. Expression contenant un opérateur

2.1.2.1. Opérateur associant un slot à une classe

1. expr = ISofClass(slotname, classname)

Attribut class-depend
class-depend(expr) =
[(slotname, classname)]
si slotname ∈ classname

undefined
sinon

2. expr = ISofClass(slotname, variable)

Attribut class-depend
class-depend(expr) =
[(slotname, Clref)]
si ∃ ‹variable, typevar, Clref› ∈ inh-tabVar(expr)
& si slotname ∈ Clref

undefined
sinon

3. expr = ISofClass(slotname, expr2)

Attribut class-depend
class-depend(expr) =
[(slotname, class-refer(expr2))] U class-depend(expr2)
si slotname ∈ class-refer(expr2)

undefined
sinon

2.1.2.2. Opérateur formant un ensemble

1. expr = SET(expr1)

Attribut class-depend
class-depend(expr) = class-depend($expr_1$)

2.1.2.3. Opérateur posant une condition sur un objet

1. expr = COND(expr1, expr2)

où

$expr_1$ est un nom de classe

Attribut class-depend
class-depend(expr) = [$expr_1$] U class-depend($expr_2$)

2. expr = COND(expr1, expr2)

où
$expr_1$ est une variable

Attribut class-depend
class-depend(expr) = class-depend($expr_1$)
class-depend($expr_1$) =
[ClrefInh] U class-depend($expr_2$)
si ∃ ‹$expr_1$, typevarInh, ClrefInh› ∈ inh-tabVar(expr)
& si (typevar ≠ undet)
& si (∄ ‹$expr_1$, typevar, Clref› ∈ tabVar($expr_2$)

[Clref] U class-depend($expr_2$)
si (∄ ‹$expr_1$,typevarInh,ClrefInh› ∈ inh-tabVar(expr))
& si (∃ ‹$expr_1$, typevar, Clref› ∈ tabVar($expr_2$))
& si (typevar ≠ undet)

undefined sinon

2.1.2.4. Opérateurs logiques

1. expr = NOT(expr1)

Attribut class-depend
class-depend(expr) = class-depend($expr_1$)

2. expr = AND(expr1, expr2)

Attribut class-depend
class-depend(expr) = class-depend($expr_1$) U class-depend($expr_2$)

3. expr = OR(expr1, expr2)

Attribut class-depend
class-depend(expr) = class-depend($expr_1$) U class-depend($expr_2$)

4. expr = EXIST-Member(expr1, expr2, expr3), FORALL-Member(expr1, expr2, expr3)

Attribut class-depend
class-depend(expr) = class-depend($expr_2$) U class-depend($expr_3$)

5. expr = EXIST-ISIN(expr1, expr2, expr3), FORALL-ISIN(expr1, expr2, expr3)

Attribut class-depend
class-depend(expr) = class-depend($expr_2$) U class-depend($expr_3$)

6. expr = EXIST-Included(expr1, expr2, expr3), FORALL-Included(expr1, expr2, expr3)

Attribut class-depend
class-depend(expr) = class-depend($expr_2$) U class-depend($expr_3$)

2.1.2.5. Opérateurs de comparaison

1. expr = Cop(expr1, expr2)

où

- Cop est l'un des opérateurs : EQ, NE, GT, GE, ST, SE

Attribut class-depend
class-depend(expr) = class-depend($expr_1$) U class-depend($expr_2$)

2.1.2.6. Opérateurs ensemblistes

1. expr = setOp(expr1, expr2)

où

- setOp est l'opérateur : Member, Included, ISIN, EqSet

Attribut class-depend
class-depend(expr) = class-depend($expr_1$) U class-depend($expr_2$)

2.1.2.7. Opérateurs définissant des fonctions agrégées

1. expr = FUNCT(expr1)

où

- FUNCT est l'un des opérateurs :

  AVG, MIN, MAX, SUM, COUNT

Attribut class-depend
class-depend(expr) = class-depend($expr_1$)

2.1.2.8. Opérateurs arithmétiques

1. expr = AOp(expr1, expr2)

où

- AOp est l'un des opérateurs : PLUS, MINUS, TIMES, DIV

Attribut class-depend
class-depend(expr) = class-depend($expr_1$) U class-depend($expr_2$)

2.2. Relations de dépendance d'une définition de slot

Soit une définition de classe Cl contenant un slot S.

Le slot S slot possède une facette (def, PCLexpression), pour laquelle on désire obtenir les relations de dépendance.

expr = PCLexpression

Attribut class-depend
class-depend(expr) = class-depend(PCLexpression)

2.3. Relations de dépendance d'une requête

Soit une requête, pour laquelle on désire obtenir les relations de dépendance.

expr = PCLquery

Attribut class-depend
class-depend(expr) = class-depend(PCLquery)

Grammaire attribuée définissant la sémantique

Annexe 7

1. Introduction

Ce document présente la définition de la sémantique du langage PCL.

Le premier point constitue la description de la grammaire attribuée définissant la sémantique d'une expression, tandis que le deuxième point constitue la définition de la grammaire attribuée pour chaque forme d'expression de base.

2. Description de la grammaire

La description de la grammaire attribuée pour la sémantique consiste en la définition de chacun de ses attributs.

Un attribut de la grammaire est une fonction définie par :

- le nom de la fonction (par convention, devenu le nom de l'attribut),
- l'ensemble de départ
- l'ensemble d'arrivée

Un dernier point décrira de manière succincte la sémantique de chaque attribut.

2.1. Attributs utilisés

1. Attribut eval

eval : Expr ----------> P(INS) + P(P(INS)) + P(B) + {error}

exp ~~~~~~~~> eval(exp)

2. Attribut inh-ins

inh-ins : Expr ----------> (INS+P(INS)) + {no-value}

exp ~~~~~~~~> inh-ins(exp) = setofinst

3. Attribut inh-Table

inh-Table : Expr ----------> TableofInst + {empty}

exp ~~~~~~~~> inh-Table(exp) = setofinst

4. Attribut Table

Table : Expr ----------> (INS+P(INS)) X (TableofInst+ {empty})

exp ~~~~~~~~> Table(exp) = (ins, setofinst)

5. Attribut denote

denote : Expr ----------> {decl, val }

exp ~~~~~~~~> denote(exp)

6. Attribut insForEval

insForEval : Expr ----------> INS + {no-value}

exp ~~~~~~~~> insForEval(exp)

Où les ensembles suivants sont :

- Expr = { expressions }
- B = { TRUE, FALSE }
- INS = { identifiants d'instances (valeurs du slot implicite "identifier") de toutes les classes du modèle, et toutes les occurences possibles de t-uples}
- TableofInst = { ‹x, type(x), y› où x est une variable ou un nom de classe, et y ∈ INS }

2.2. Sémantique des attributs

1. Attribut eval

La fonction **eval** associe à chaque expression exp une valeur eval(exp) qui est :

eval(exp) =

- un ensemble de valeurs

  où chaque valeur est une valeur possible de l'expression, cette valeur étant le résultat de l'évaluation de l'expression pour un jeu d'instanciations particulier pour les variables et noms de classes (une valeur pouvant être un ensemble).

- error

  sinon.

2. Attributs inh-ins, inh-Table, et Table

La fonction **inh-ins** associe à chaque expression exp une valeur inh-ins(exp) qui est :

inh-ins(exp) =

- une valeur ins

  où ins est un identifiant d'instance associé à une valeur setofinst définie par la fonction inh-Table, et où ins permet de caractériser cet ensemble.

- no-value

  sinon (si eval(expr) = error pour une expression plus générale, ou si la valeur setofinst est { }).

La fonction **inh-Table** associe à chaque expression exp une valeur inh-Table(exp) qui est :

inh-Table(exp) =

- une valeur setofinst

  où setofinst est un jeu d'instanciations, un ensemble d'éléments ‹x,type(x),insforx› pour chaque variable ou nom de classe apparaissant dans exp (variables ou noms de classes pour lesquels une valeur est déjà fixée).

- empty

  sinon (si eval(expr) = error pour une expression plus générale).

La fonction **Table** associe à chaque expression exp une valeur Table(exp) qui est :

Table(exp) =

- un couple (ins, setofinst)

  où ins est une des valeurs du résultat de l'évaluation de exp (ins ∈ eval(exp)), et setofinst est l'ensemble d'éléments ‹x,type(x),insforx› pour chaque variable ou nom de classe apparaissant dans exp.

- empty

sinon (si eval(expr) = error).

3. Attribut denote

La fonction **denote** associe à chaque expression exp une valeur denote(exp) qui caractérise le type d'objet désigné par exp :

- val pour une expression désignant une valeur
- decl pour une expression désignant une déclaration de type

4. Attribut insForEval

La fonction **insForEval** associe à chaque expression exp une valeur insForEval(exp) qui est :

insForEval(exp) =

- un identifiant d'instance (valeur du slot implicite **identifier**)

  quand une instance doit (ou peut) être spécifiée pour définir la sémantique de exp, cette instance étant une instance pour laquelle on désire évaluer l'expression.

- no-value

  sinon.

3. Grammaire attribuée

La sémantique d'une expression est constituée de la définition de la sémantique de chaque expression de base.

De cette manière, pour chaque forme particulière d'une expression, on définit la valeur des attributs de la grammaire attribuée.

Un dernier point précisera la sémantique d'une expression dans les cas particuliers de requête ou d'expression définissant un slot.

3.1. Sémantique d'une expression

Les pages suivantes constituent la définition de la sémantique d'une expression selon la forme de cette expression.

3.1.1. Expression de base

1. expr = constante

Attribut eval

eval(expr) = selon les cas :

{ constante }	si inhTable(expr) ≠ empty
error	sinon

Attributs inh-Table, inh-ins et Table

inh-ins(constante) = inh-ins(expr)

inh-Table(constante) = inh-Table(expr)

Table(expr) = selon les cas :

- { (inh-ins(expr), inh-Table(expr)) }
  si inh-Table(expr) ≠ empty
- empty
  sinon

Attribut denote

denote(constante) = denote(expr) = val

Attribut insForEval

insForEval(constante) = insForEval(expr)

2. expr = classname

Attribut eval

eval(expr) = selon les cas :

- { insForEval(classname) }
  si insForEval(classname) ∈ E[classname]
  & insForEval(classname) ≠ no-value
  & denote(expr) = val

- { insval }
  si insForEval(classname) ∉ E[classname]
  & denote(expr) = val
  & ∃ ‹classname,type(classname),insval› ∈inh-Table(expr)

- { ins, ∀ ins ∈ E[Classname] }
  si denote(expr) = val
  & ∄ ‹classname, type(classname), insval› ∈ inh-Table(classname)

- E[classname]
  si denote(expr) = decl

- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins(classname) = inh-ins(expr)
inh-Table(classname) = inh-Table(expr)
Table(expr) = selon les cas :

- { (x, inst),
  ∀ x ∈ eval(expr)
  & inst =(inh-Table(expr)U{‹classname,type(classname),x›}
  }
  si eval(expr) ≠ error
- empty
  sinon

Attribut denote

denote(classname) = denote(expr)

Attribut insForEval

insForEval(classname) = insForEval(expr)

3. expr = variable

Attribut eval

eval(expr) = selon les cas :

- { insval }
  si ∃ ‹variable, typevar ,insval› ∈ inh-Table(expr)
- insForEval(variable)
  si type(variable) = type (insForEval(variable))
- selon les cas :
  si ∄ ‹variable, typevar ,insval› ∈ inh-Table(expr) & inh-Table(expr) ≠ empty
  * eval(classname) si type(expr) = classname
  * {x, ∀ x entier} si type(expr) = integer
  * {x, ∀ x réel} si type(expr) = real
  * {x, ∀ x string} si type(expr) = string
  * T si type(expr) = tuple
  * P(X) si type(expr) = set(y),
    où y est l'un des 5 types précédents,
    et X l'ensemble correspondant.
- error
  sinon

Note :

L'ensemble T est défini par T = { tous les t-uples composés d'éléments ∈ INS }

Attributs inh-Table, inh-ins et Table

inh-ins(variable) = inh-ins(expr)

inh-Table(variable) = inh-Table(expr)

Table(expr) = selon les cas :

- { (x, inst), ∀ x ∈ eval(expr)
  & inst = (inh-Table(expr) U {‹variable,type(variable), x›}
  }
  si eval(expr) ≠ error
- empty
  sinon

Attribut denote

denote(variable) = denote(expr)

Attribut insForEval

insForEval(variable) = insForEval(expr)

4. expr = slotname

Attribut eval
eval(expr) = { slotname(insForEval(slotname)) }
où slotname( ) est la fonction associée au slot de nom slotname, comme définie dans le modèle conceptuel.

Attributs inh-Table, inh-ins et Table
inh-ins(slotname) = inh-ins(expr)
inh-Table(slotname) = inh-Table(expr)
Table(expr) = inh-Table(slotname)

Attribut denote
denote(slotname) = denote(expr)

Attribut insForEval
insForEval(slotname) = insForEval(expr)

5. expr = (t1, ..., tn)

Attribut eval

eval(expr) = selon les cas :

- { (val_1, ... , val_n), ∀ (val_1, ... , val_n) où
  $val_1 \in eval(t_1)$
  $val_2 \in eval(t_2)$ for which inh-Table(t_2) = $inst_1$, &
  (val_1,$inst_1$) ∈ Table(t_1)
  .....
  $val_n \in eval(t_n)$ for which inh-Table(t_n) = $inst_{n-1}$, &
  (val_{n-1},$inst_{n-1}$) ∈ Table(t_{n-1})
  }
  si ∀ i : eval(t_i) ≠ error, ≠ { } , inh-Table(expr) ≠ empty
- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins(t_1) = inh-ins(expr)
inh-Table(t_1) = inh-Table(expr)
∀ i : 1 < i<= n , ∀ x ∈ eval(t_{i-1}), (x,$inst_{n-1}$) ∈ Table(t_{i-1}) :
inh-ins(t_i) = x
inh-Table(t_i) = $inst_{i-1}$
Table(expr) = selon les cas :

- { ((val_1, ... , val_n), $inst_n$),
  ∀ (val_1, ... , val_n) ∈ eval(expr)
  & (val_n,$inst_n$) ∈ Table(t_n)
  }
  si eval(expr) ≠ error
- empty
  sinon

Attribut denote

∀ i : 1 <= i<= n :
denote(t_i) = denote(expr) = val

Attribut insForEval

∀ i : 1 <= i<= n :
insForEval(t_i) = insForEval(expr)

6. expr = [c1, ..., cn]

Attribut eval

eval(expr) = selon les cas :

- { {eval(c_1), ... , eval(c_n)} }
  si ∀ i : 1 <= i <= n : eval(c_i) ≠ error
- error
  sinon

Attributs inh-Table, inh-ins et Table

∀ i : 1 <= i<= n :

inh-ins(c_i) = inh-ins(expr)

inh-Table(c_i) = inh-Table(expr)

Table(expr) = selon les cas :

- { ((val_1, ... , val_n), inst), ∀ val_i ∈ eval(c_i) & inst = inh-Table(expr)
  }
  si eval(expr) ≠ error
- empty
  sinon

Attribut denote

∀ i : 1 <= i<= n :

denote(c_i) = denote(expr) = val

Attribut insForEval

∀ i : 1 <= i<= n :

insForEval(c_i) = insForEval(expr)

3.1.2. Expression contenant un opérateur

1. Opérateur associant un slot à une classe

1. expr = ISofClass(slotname, $expr_2$)

Attribut eval

eval(expr) = selon les cas :

- { eval(slotname), ∀ eval(slotname) ≠ error
  où
  inh-ins(slotname) = x,
  inh-Table(slotname) = inst,
  (x, inst) ∈ Table(t_2), ∀ x ∈ eval(t_2)
  }
  si eval($expr_2$) ≠ error
- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins($expr_2$) = inh-ins(expr)

inh-Table($expr_2$) = inh-Table(expr)

∀ x ∈ eval($expr_2$), (x,inst) ∈ Table($expr_2$) :
inh-ins(slotname) = x
inh-Table(slotname) = inst

Table(expr) = selon les cas :

- $\bigcup \left[\text{Table(slotname)}\right] \Big|_{\forall (x,inst) \in \text{Table(expr2)}}$
  ∀ Table(slotname) où
  inh-Table(slotname) = inst,
  inh-ins(slotname) = x,
  eval(slotname) ≠ empty
  si eval(expr) ≠ error
- empty
  sinon

Attribut denote

denote(slotname) = denote(expr)

denote($expr_2$) = denote(expr)

Attribut insForEval

insForEval(slotname) = x, ∀ x ∈ eval($expr_2$)

insForEval($expr_2$) = insForEval(expr)

2. Opérateur formant un ensemble

1. expr = SET(expr1)

Attribut eval

eval(expr) = selon les cas :

- **{** {x, ∀ x ∈ eval($expr_1$) & (x,Y) ∈ Table($expr_1$)},
  ∀ Z = Y \ { ‹$expr_1$, $type_1$, x› } **}**
  si eval($expr_1$) ≠ error, inh-Table(expr) ≠ empty,
  denote(expr) = val
- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins($expr_1$) = inh-ins(expr)

inh-Table($expr_1$) = inh-Table(expr)

Table(expr) = selon les cas :

- { (inst-set, Y), ∀ inst-set ∈ eval(expr)
  & ∀ ((val, Y) ∈ Table($expr_1$) ∀ val ∈ inst-set)
  }
  si eval($expr_1$) ≠ error, inh-Table(expr) ≠ empty,
  denote(expr) = val
- empty
  sinon

Attribut denote

denote($expr_1$) = denote(expr)

Attribut insForEval

insForEval($expr_1$) = insForEval(expr)

3. Opérateur posant une condition sur un objet

1. expr = COND(expr1, expr2)

Attribut eval

eval(expr) = selon les cas :

- { x, ∀ x ∈ eval($expr_1$)
  où
  (x, setinst) ∈ Table($expr_1$)
  inh-ins($expr_2$) = x
  inh-Table($expr_2$) = setinst
  eval($expr_2$) = TRUE
  }
  si eval($expr_1$) ≠ error, eval($expr_2$) ≠ error
- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins($expr_1$) = inh-ins(expr)
inh-Table($expr_1$) = inh-Table(expr)
∀ x ∈ eval($expr_1$), (x,inst) ∈ Table($expr_1$) :
inh-ins($expr_2$) = x
inh-Table($expr_2$) = inst
Table(expr) = selon les cas :

- **{** (x, setinst**)**, ∀ x ∈ eval(expr) & ∀ (x, setinst) ∈ **S**
  où
  S = $\bigcup \left[\text{Table}(expr_2)\right] \Big|_{\forall (x,inst) \in \text{Table}(expr1)}$
  ∀ Table($expr_2$) où
  inh-Table($expr_2$) = inst,
  inh-ins($expr_2$) = x,
  Table($expr_2$) ≠ empty
  }
  si eval(expr) ≠ error
- empty
  sinon

Attribut denote

denote($expr_1$) = denote(expr)
denote($expr_2$) = denote(expr)

Attribut insForEval

insForEval($expr_1$) = insForEval(expr)
insForEval($expr_2$) = insForEval(expr)

4. Opérateurs logiques

1. expr = NOT(expr1)

Attribut eval

eval(expr) = selon les cas :

- TRUE
  si eval($expr_1$) = FALSE
- FALSE
  si eval($expr_1$) = TRUE
- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins($expr_1$) = inh-ins(expr)
inh-Table($expr_1$) = inh-Table(expr)
Table(expr) = Table($expr_1$)

Attribut denote

denote($expr_1$) = denote(expr)

Attribut insForEval

insForEval($expr_1$) = insForEval(expr)

2. expr = AND(expr1, expr2)

Attribut eval

eval(expr) = selon les cas :

- TRUE
  si eval($expr_1$) = TRUE & eval($expr_2$) = TRUE
- FALSE
  si eval($expr_1$) ≠ error & eval($expr_2$) ≠ error
  & (eval($expr_1$) = FALSE ou eval($expr_2$) = FALSE)
- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins($expr_1$) = inh-ins(expr)

inh-Table($expr_1$) = inh-Table(expr)

∀ (x,inst) ∈ Table($expr_1$) :

  inh-ins($expr_2$) = x

  inh-Table($expr_2$) = inst

Table(expr) = selon les cas :

- $\bigcup \left[\text{Table}(expr_2) \right] \Big|_{\forall (x,inst) \in \text{Table}(expr1)}$
  ∀ Table($expr_2$) où
  inh-Table($expr_2$) = inst,
  inh-ins($expr_2$) = x,
  Table($expr_2$) ≠ empty
  si eval(expr) = TRUE
- { }
  si eval(expr) = FALSE
- empty
  sinon

Attribut denote

denote($expr_1$) = denote(expr)

denote($expr_2$) = denote(expr)

Attribut insForEval

insForEval($expr_1$) = insForEval(expr)

insForEval($expr_2$) = insForEval(expr)

3. expr = OR(expr1, expr2)

Attribut eval

eval(expr) = selon les cas :

- TRUE
  si eval($expr_1$) = TRUE ou eval($expr_2$) = TRUE)
- FALSE
  si eval($expr_1$) = FALSE & eval($expr_2$) = FALSE
- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins($expr_1$) = inh-ins(expr)
inh-Table($expr_1$) = inh-Table(expr)
inh-ins($expr_2$) = inh-ins(expr)
inh-Table($expr_2$) = inh-Table(expr)
Table(expr) = selon les cas :

- Table($expr_1$) U Table($expr_2$)
  si eval(expr) = TRUE
- { }
  si eval(expr) = FALSE
- empty
  sinon

Attribut denote

denote($expr_1$) = denote(expr)
denote($expr_2$) = denote(expr)

Attribut insForEval

insForEval($expr_1$) = insForEval(expr)
insForEval($expr_2$) = insForEval(expr)

4. expr = EXIST-Member(expr1, expr2, expr3)
expr = EXIST-ISIN(expr1, expr2, expr3)
expr = EXIST-Included(expr1, expr2, expr3)

Attribut eval

eval(expr) = selon les cas :

- TRUE
  si eval($expr_1$) ≠ error & eval($expr_1$) ≠ { }
  & eval($expr_2$) ≠ error & eval($expr_2$) ≠ { }
  & ∃ x ∈ eval($expr_1$)
  & ∃ Y ∈ eval($expr_2$)
  & x ∈ Y
  :
  (inh-ins($expr_3$) = inh-ins(expr)
  & inh-Table($expr_3$) = instX U instY
  & (x, instX) ∈ Table($expr_1$)
  & (Y, instY) ∈ Table($expr_2$))
  ⟹ eval($expr_3$) = TRUE
- FALSE
  si eval($expr_1$) ≠ error & eval($expr_1$) ≠ { }
  & eval($expr_2$) ≠ error & eval($expr_2$) ≠ { }
  & ∀ x ∈ eval($expr_1$) & ∀ Y ∈ eval($expr_2$) & x ∈ Y
  :
  (inh-ins($expr_3$) = inh-ins(expr)
  & inh-Table($expr_3$) = instX U instY
  & (x, instX) ∈ Table($expr_1$)
  & (Y, instY) ∈ Table($expr_2$))
  ⟹ eval($expr_3$) = FALSE
  ou si eval($expr_1$) ≠ error
  & eval($expr_2$) ≠ error
  & ∀ x ∈ eval($expr_1$)
  & ∀ Y ∈ eval($expr_2$)
  :
  x ∉ Y
- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins($expr_1$) = inh-ins(expr)
inh-Table($expr_1$) = inh-Table(expr)
inh-ins($expr_2$) = inh-ins(expr)
inh-Table($expr_2$) = inh-Table(expr)
∀ (x,instX) ∈ Table($expr_1$), (y,instY) ∈ Table($expr_2$) :
inh-ins($expr_3$) = inh-ins(expr)
inh-Table($expr_3$) = instX U instY
Table(expr) = Table($expr_3$)

Attribut denote

denote($expr_1$) = denote(expr)
denote($expr_2$) = denote(expr)
denote($expr_3$) = denote(expr)

Attribut insForEval

insForEval($expr_1$) = insForEval(expr)
insForEval($expr_2$) = insForEval(expr)
insForEval($expr_3$) = insForEval(expr)

5. expr = FORALL-Member(expr1, expr2, expr3)
expr = FORALL-ISIN(expr1, expr2, expr3)
expr = FORALL-Included(expr1, expr2, expr3)

Attribut eval

eval(expr) = selon les cas :

- TRUE
  si eval($expr_1$) ≠ error & eval($expr_1$) ≠ { }
  & eval($expr_2$) ≠ error & eval($expr_2$) ≠ { }
  & ∃ x ∈ eval($expr_1$)
  & ∃ Y ∈ eval($expr_2$)
  & x ∈ Y
  :
  (inh-ins($expr_3$) = inh-ins(expr)
  & inh-Table($expr_3$) = instX U instY
  & (x, instX) ∈ Table($expr_1$)
  & (Y, instY) ∈ Table($expr_2$))
  ⟹ eval($expr_3$) = TRUE
- FALSE
  si eval($expr_1$) ≠ error & eval($expr_1$) ≠ { }
  & eval($expr_2$) ≠ error & eval($expr_2$) ≠ { }
  & ∀ x ∈ eval($expr_1$) **&** ∀ Y ∈ eval($expr_2$) **&** x ∈ Y
  :
  (inh-ins($expr_3$) = inh-ins(expr)
  & inh-Table($expr_3$) = instX U instY
  & (x, instX) ∈ Table($expr_1$)
  & (Y, instY) ∈ Table($expr_2$))
  ⟹ eval($expr_3$) = FALSE
  ou si eval($expr_1$) ≠ error
  & eval($expr_2$) ≠ error
  & ∀ x ∈ eval($expr_1$)
  & ∀ Y ∈ eval($expr_2$)
  :
  x ∉ Y
- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins($expr_1$) = inh-ins(expr)
inh-Table($expr_1$) = inh-Table(expr)
inh-ins($expr_2$) = inh-ins(expr)
inh-Table($expr_2$) = inh-Table(expr)
∀ (x,instX) ∈ Table($expr_1$), (y,instY) ∈ Table($expr_2$) :
inh-ins($expr_3$) = inh-ins(expr)
inh-Table($expr_3$) = instX U instY
Table(expr) = Table($expr_3$)

Attribut denote

denote($expr_1$) = denote(expr)
denote($expr_2$) = denote(expr)
denote($expr_3$) = denote(expr)

Attribut insForEval

insForEval($expr_1$) = insForEval(expr)
insForEval($expr_2$) = insForEval(expr)
insForEval($expr_3$) = insForEval(expr)

5. Opérateurs de comparaison

1. expr = EQ(expr1, expr2)

Attribut eval

eval(expr) = selon les cas :

- TRUE
  si eval($expr_1$) ≠ error & eval($expr_1$) ≠ { }
  & eval($expr_2$) ≠ error & eval($expr_2$) ≠ { }
  & ∃ x ∈ eval($expr_1$)
  :
  (inh-ins($expr_2$) = inh-ins(expr)
  & inh-Table($expr_2$) = instX
  & (x, instX) ∈ Table($expr_1$)
)
  ⟹ ∃ y ∈ eval($expr_2$) : x = y
- FALSE
  si eval($expr_1$) ≠ error
  & eval($expr_2$) ≠ error
  & ∀ x ∈ eval($expr_1$)
  :
  (inh-ins($expr_2$) = inh-ins(expr)
  & inh-Table($expr_2$) = instX
  & (x, instX) ∈ Table($expr_1$)
)
  ⟹∀ y ∈ eval($expr_2$) : x ≠ y
- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins($expr_1$) = inh-ins(expr)
inh-Table($expr_1$) = inh-Table(expr)
∀ (x,instX) ∈ Table($expr_1$) :
  inh-ins($expr_2$) = x
  inh-Table($expr_2$) = instX

Table(expr) = selon les cas :

- $U[U[\{(A, B)\}]\,|_{\forall y \in eval(expr2)}]\,|_{\forall x \in eval(expr1)}$
  ∀ (A, B) où
    A = inh-ins(expr), B = instY
    (x, instX) ∈ Table($expr_1$)
    inh-ins($expr_2$) = x, inh-Table($expr_2$) = instX
    (y, instY) ∈ Table($expr_2$)
    x = y
  si eval(expr) = TRUE
- { }
  si eval(expr) = FALSE
- empty
  sinon

Attribut denote

$denote(expr_1) = denote(expr)$

$denote(expr_2) = denote(expr)$

Attribut insForEval

$insForEval(expr_1) = insForEval(expr)$

$insForEval(expr_2) = insForEval(expr)$

2. expr = NE(expr1, expr2)

Attribut eval

eval(expr) = selon les cas :

- TRUE
  si eval($expr_1$) ≠ error & eval($expr_1$) ≠ { }
  & eval($expr_2$) ≠ error & eval($expr_2$) ≠ { }
  & ∃ x ∈ eval($expr_1$)
  :
  (inh-ins($expr_2$) = inh-ins(expr)
  & inh-Table($expr_2$) = instX
  & (x, instX) ∈ Table($expr_1$)
)
  ⟹ ∃ y ∈ eval($expr_2$) : x ≠ y
- FALSE
  si eval($expr_1$) ≠ error
  & eval($expr_2$) ≠ error
  & ∀ x ∈ eval($expr_1$)
  :
  (inh-ins($expr_2$) = inh-ins(expr)
  & inh-Table($expr_2$) = instX
  & (x, instX) ∈ Table($expr_1$)
)
  ⟹∀ y ∈ eval($expr_2$) : x = y
- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins($expr_1$) = inh-ins(expr)
inh-Table($expr_1$) = inh-Table(expr)
∀ (x,instX) ∈ Table($expr_1$) :
  inh-ins($expr_2$) = x
  inh-Table($expr_2$) = instX
Table(expr) = selon les cas :

- $\cup[\cup[\{(A, B)\}]\,|_{\forall y \in eval(expr2)}]\,|_{\forall x \in eval(expr1)}$
  ∀ (A, B) où
    A = inh-ins(expr)
    B = instY
    (x, instX) ∈ Table($expr_1$)
    inh-ins($expr_2$) = x,
    inh-Table($expr_2$) = instX
    (y, instY) ∈ Table($expr_2$)
    x ≠ y
  si eval(expr) = TRUE
- { }
  si eval(expr) = FALSE
- empty
  sinon

Attribut denote

denote($expr_1$) = denote(expr)
denote($expr_2$) = denote(expr)

Attribut insForEval

insForEval($expr_1$) = insForEval(expr)
insForEval($expr_2$) = insForEval(expr)

3. expr = GT(expr1, expr2)

Attribut eval

eval(expr) = selon les cas :

- TRUE
  si eval($expr_1$) ≠ error & eval($expr_1$) ≠ { }
  & eval($expr_2$) ≠ error & eval($expr_2$) ≠ { }
  & ∃ x ∈ eval($expr_1$)
  :
  (inh-ins($expr_2$) = inh-ins(expr)
  & inh-Table($expr_2$) = instX
  & (x, instX) ∈ Table($expr_1$)
)
  ⟹ ∃ y ∈ eval($expr_2$) : x > y
- FALSE
  si eval($expr_1$) ≠ error
  & eval($expr_2$) ≠ error
  & ∀ x ∈ eval($expr_1$)
  :
  (inh-ins($expr_2$) = inh-ins(expr)
  & inh-Table($expr_2$) = instX
  & (x, instX) ∈ Table($expr_1$)
)
  ⟹ ∀ y ∈ eval($expr_2$) : x <= y
- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins($expr_1$) = inh-ins(expr)
inh-Table($expr_1$) = inh-Table(expr)
∀ (x,instX) ∈ Table($expr_1$) :
  inh-ins($expr_2$) = x
  inh-Table($expr_2$) = instX

Table(expr) = selon les cas :

- $U[U[\{(A, B)\}]\,|_{\forall y \in eval(expr2)}]\,|_{\forall x \in eval(expr1)}$
  ∀ (A, B) où
  A = inh-ins(expr)
  B = instY
  (x, instX) ∈ Table($expr_1$)
  inh-ins($expr_2$) = x,
  inh-Table($expr_2$) = instX
  (y, instY) ∈ Table($expr_2$)
  x > y
  si eval(expr) = TRUE
- { }
  si eval(expr) = FALSE
- empty
  sinon

Attribut denote

denote($expr_1$) = denote(expr)

denote($expr_2$) = denote(expr)

Attribut insForEval

insForEval($expr_1$) = insForEval(expr)

insForEval($expr_2$) = insForEval(expr)

4. expr = GE(expr1, expr2)

Attribut eval

eval(expr) = selon les cas :

- TRUE
  si eval($expr_1$) ≠ error & eval($expr_1$) ≠ { }
  & eval($expr_2$) ≠ error & eval($expr_2$) ≠ { }
  & ∃ x ∈ eval($expr_1$)
  :
  (inh-ins($expr_2$) = inh-ins(expr)
  & inh-Table($expr_2$) = instX
  & (x, instX) ∈ Table($expr_1$)
)
  ⟹ ∃ y ∈ eval($expr_2$) : x >= y
- FALSE
  si eval($expr_1$) ≠ error
  & eval($expr_2$) ≠ error
  & ∀ x ∈ eval($expr_1$)
  :
  (inh-ins($expr_2$) = inh-ins(expr)
  & inh-Table($expr_2$) = instX
  & (x, instX) ∈ Table($expr_1$)
)
  ⟹ ∀ y ∈ eval($expr_2$) : x < y
- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins($expr_1$) = inh-ins(expr)
inh-Table($expr_1$) = inh-Table(expr)
∀ (x,instX) ∈ Table($expr_1$) :
  inh-ins($expr_2$) = x
  inh-Table($expr_2$) = instX
Table(expr) = selon les cas :

- $\bigcup\left[\bigcup\left[\{(A, B)\}\right]\Big|_{\forall y \in eval(expr2)}\right]\Big|_{\forall x \in eval(expr1)}$
  ∀ (A, B) où
  A = inh-ins(expr)
  B = instY
  (x, instX) ∈ Table($expr_1$)
  inh-ins($expr_2$) = x,
  inh-Table($expr_2$) = instX
  (y, instY) ∈ Table($expr_2$)
  x >= y
  si eval(expr) = TRUE
- { }
  si eval(expr) = FALSE
- empty
  sinon

Attribut denote

denote($expr_1$) = denote(expr)

denote($expr_2$) = denote(expr)

Attribut insForEval

insForEval($expr_1$) = insForEval(expr)

insForEval($expr_2$) = insForEval(expr)

5. expr = ST(expr1, expr2)

Attribut eval

eval(expr) = selon les cas :

- TRUE
  si eval($expr_1$) ≠ error & eval($expr_1$) ≠ { }
  & eval($expr_2$) ≠ error & eval($expr_2$) ≠ { }
  & ∃ x ∈ eval($expr_1$)
  :
  (inh-ins($expr_2$) = inh-ins(expr)
  & inh-Table($expr_2$) = instX
  & (x, instX) ∈ Table($expr_1$)
)
  ⟹ ∃ y ∈ eval($expr_2$) : x < y
- FALSE
  si eval($expr_1$) ≠ error
  & eval($expr_2$) ≠ error
  & ∀ x ∈ eval($expr_1$)
  :
  (inh-ins($expr_2$) = inh-ins(expr)
  & inh-Table($expr_2$) = instX
  & (x, instX) ∈ Table($expr_1$)
)
  ⟹∀ y ∈ eval($expr_2$) : x >= y
- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins($expr_1$) = inh-ins(expr)
inh-Table($expr_1$) = inh-Table(expr)
∀ (x,instX) ∈ Table($expr_1$) :
  inh-ins($expr_2$) = x
  inh-Table($expr_2$) = instX
Table(expr) = selon les cas :

- $\cup[\cup[\{(A, B)\}]\,|_{\forall y \in eval(expr2)}]\,|_{\forall x \in eval(expr1)}$
  ∀ (A, B) où
  A = inh-ins(expr)
  B = instY
  (x, instX) ∈ Table($expr_1$)
  inh-ins($expr_2$) = x,
  inh-Table($expr_2$) = instX
  (y, instY) ∈ Table($expr_2$)
  x < y
  si eval(expr) = TRUE
- { }
  si eval(expr) = FALSE
- empty
  sinon

Attribut denote

denote($expr_1$) = denote(expr)

denote($expr_2$) = denote(expr)

Attribut insForEval

insForEval($expr_1$) = insForEval(expr)

insForEval($expr_2$) = insForEval(expr)

6. expr = SE(expr1, expr2)

Attribut eval

eval(expr) = selon les cas :

- TRUE
  si eval($expr_1$) ≠ error & eval($expr_1$) ≠ { }
  & eval($expr_2$) ≠ error & eval($expr_2$) ≠ { }
  & ∃ x ∈ eval($expr_1$)
  :
  (inh-ins($expr_2$) = inh-ins(expr)
  & inh-Table($expr_2$) = instX
  & (x, instX) ∈ Table($expr_1$)
)
  ⟹ ∃ y ∈ eval($expr_2$) : x <= y
- FALSE
  si eval($expr_1$) ≠ error
  & eval($expr_2$) ≠ error
  & ∀ x ∈ eval($expr_1$)
  :
  (inh-ins($expr_2$) = inh-ins(expr)
  & inh-Table($expr_2$) = instX
  & (x, instX) ∈ Table($expr_1$)
)
  ⟹ ∀ y ∈ eval($expr_2$) : x > y
- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins($expr_1$) = inh-ins(expr)
inh-Table($expr_1$) = inh-Table(expr)
∀ (x,instX) ∈ Table($expr_1$) :
  inh-ins($expr_2$) = x
  inh-Table($expr_2$) = instX

Table(expr) = selon les cas :

- $U[U[\{(A, B)\}]\,|_{\forall y \in eval(expr2)}]\,|_{\forall x \in eval(expr1)}$
  ∀ (A, B) où
    A = inh-ins(expr)
    B = instY
    (x, instX) ∈ Table($expr_1$)
    inh-ins($expr_2$) = x,
    inh-Table($expr_2$) = instX
    (y, instY) ∈ Table($expr_2$)
    x <= y
  si eval(expr) = TRUE
- { }
  si eval(expr) = FALSE
- empty
  sinon

Attribut denote

denote($expr_1$) = denote(expr)

denote($expr_2$) = denote(expr)

Attribut insForEval

insForEval($expr_1$) = insForEval(expr)

insForEval($expr_2$) = insForEval(expr)

6. Opérateurs ensemblistes

1. expr = Member(expr1, expr2)

Attribut eval

eval(expr) = selon les cas :

- TRUE
  si eval($expr_1$) ≠ error & eval($expr_1$) ≠ { }
  & eval($expr_2$) ≠ error & eval($expr_2$) ≠ { }
  & ∃ x ∈ eval($expr_1$)
  :
  (inh-ins($expr_2$) = x
  & inh-Table($expr_2$) = instX
  & (x, instX) ∈ Table($expr_1$)
)
  ⟹ ∃ y ∈ eval($expr_2$) : x ∈ y
- FALSE
  si eval($expr_1$) ≠ error & eval($expr_1$) ≠ { }
  & eval($expr_2$) ≠ error
  & ∀ x ∈ eval($expr_1$)
  :
  (inh-ins($expr_2$) = x
  & inh-Table($expr_2$) = instX
  & (x, instX) ∈ Table($expr_1$)
)
  ⟹ ∀ y ∈ eval($expr_2$) : x ∉ y
- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins($expr_1$) = inh-ins(expr)
inh-Table($expr_1$) = inh-Table(expr)
∀ (x,instX) ∈ Table($expr_1$) :
  inh-ins($expr_2$) = x
  inh-Table($expr_2$) = instX
Table(expr) = selon les cas :

- $U[U[\{(A, B)\}]\,|_{\forall y \in eval(expr2)}]\,|_{\forall x \in eval(expr1)}$
  ∀ (A, B) où
  A = inh-ins(expr), B = instY
  (x, instX) ∈ Table($expr_1$)
  inh-ins($expr_2$) = x, inh-Table($expr_2$) = instX
  (y, instY) ∈ Table($expr_2$)
  x ∈ y
  si eval(expr) = TRUE
- { }
  si eval(expr) = FALSE
- empty
  sinon

Attribut denote

denote($expr_1$) = denote(expr)

denote($expr_2$) = denote(expr)

Attribut insForEval

insForEval($expr_1$) = insForEval(expr)

insForEval($expr_2$) = insForEval(expr)

2. expr = Included(expr1, expr2)

Attribut eval

eval(expr) = selon les cas :

- TRUE
  (si eval($expr_1$) ≠ error
  & eval($expr_2$) ≠ error & eval($expr_2$) ≠ { }
  & ∃ x ∈ eval($expr_1$)
  :
  (inh-ins($expr_2$) = x
  & inh-Table($expr_2$) = instX
  & (x, instX) ∈ Table($expr_1$)
)
  ⟹ ∃ y ∈ eval($expr_2$) : (∀ el ∈ x : el ∈ y) **)**
  ou si eval($expr_1$) = { } & eval($expr_2$) = { }
- FALSE
  si eval($expr_1$) ≠ error & eval($expr_1$) ≠ { }
  & eval($expr_2$) ≠ error & eval($expr_2$) ≠ { }
  & ∀ x ∈ eval($expr_1$)
  :
  (inh-ins($expr_2$) = x
  & inh-Table($expr_2$) = instX
  & (x, instX) ∈ Table($expr_1$)
)
  ⟹ ∀ y ∈ eval($expr_2$) : (∃ el ∈ x : el ∉ y)
- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins($expr_1$) = inh-ins(expr)
inh-Table($expr_1$) = inh-Table(expr)
∀ (x,instX) ∈ Table($expr_1$) :
  inh-ins($expr_2$) = x
  inh-Table($expr_2$) = instX

Table(expr) = selon les cas :

- $\cup\left[\cup\left[\{(A, B)\}\right]\Big|_{\forall y \in eval(expr2)}\right]\Big|_{\forall x \in eval(expr1)}$
  ∀ (A, B) où
  A = inh-ins(expr), B = instY
  (x, instX) ∈ Table($expr_1$)
  inh-ins($expr_2$) = x, inh-Table($expr_2$) = instX
  (y, instY) ∈ Table($expr_2$)
  (∀ el ∈ x : el ∈ y)
  si eval(expr) = TRUE
- { }
  si eval(expr) = FALSE
- empty
  sinon

Attribut denote

denote($expr_1$) = denote(expr)

denote($expr_2$) = denote(expr)

Attribut insForEval

insForEval($expr_1$) = insForEval(expr)

insForEval($expr_2$) = insForEval(expr)

3. expr = ISIN(expr1, expr2)

Attribut eval

eval(expr) = selon les cas :

- TRUE

  (si eval($expr_1$) ≠ error **&** ∃ x ∈ eval($expr_1$)
  & eval($expr_2$) ≠ error & eval($expr_2$) ≠ { }
  :

  (inh-ins($expr_2$) = x
  & inh-Table($expr_2$) = instX
  & (x, instX) ∈ Table($expr_1$) **)**

  ⟹ ∃ y ∈ eval($expr_2$) :

  (∀ el ∈ x : el ∈ y) si type(expr1) = set(z)
  x ∈ y si type(expr1) = z

) ou si eval($expr_1$) = { } & eval($expr_2$) = { }

- FALSE

  si eval($expr_1$) ≠ error & eval($expr_1$) ≠ { }
  & eval($expr_2$) ≠ error & eval($expr_2$) ≠ { }
  & ∀ x ∈ eval($expr_1$)
  :

  (inh-ins($expr_2$) = x
  & inh-Table($expr_2$) = instX
  & (x, instX) ∈ Table($expr_1$) **)**

  ⟹ ∀ y ∈ eval($expr_2$) :

  (∃ el ∈ x : el ∉ y) si type(expr1) = set(z)
  x ∉ y si type(expr1) = z

- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins($expr_1$) = inh-ins(expr)
inh-Table($expr_1$) = inh-Table(expr)
∀ (x,instX) ∈ Table($expr_1$) :
inh-ins($expr_2$) = x
inh-Table($expr_2$) = instX

Table(expr) = selon les cas :

- $U[U[\{(A, B)\}]\,|_{\forall y \in eval(expr2)}]\,|_{\forall x \in eval(expr1)}$

  ∀ (A, B) où
  A = inh-ins(expr), B = instY
  (x, instX) ∈ Table($expr_1$)
  inh-ins($expr_2$) = x, inh-Table($expr_2$) = instX
  (y, instY) ∈ Table($expr_2$)
  (∀ el ∈ x : el ∈ y) si type(expr1) = set(z)
  x ∈ y si type(expr1) = z
  si eval(expr) = TRUE

- { }
  si eval(expr) = FALSE
- empty
  sinon

Attribut denote

denote($expr_1$) = denote(expr)
denote($expr_2$) = decl

Attribut insForEval

insForEval($expr_1$) = insForEval(expr)
insForEval($expr_2$) = insForEval(expr)

4. expr = EqSet(expr1, expr2)

Attribut eval

eval(expr) = selon les cas :

- TRUE
  (**si** eval($expr_1$) ≠ error & eval($expr_1$) ≠ { }
  & eval($expr_2$) ≠ error & eval($expr_1$) ≠ { }
  & ∃ x ∈ eval($expr_1$)
  :
  (inh-ins($expr_2$) = x
  & inh-Table($expr_2$) = instX
  & (x, instX) ∈ Table($expr_1$)
)
  ⟹∃ y∈ eval($expr_2$): (∀ el ∈ x: el ∈ y) & (∀ el ∈ y: el ∈x)**)**
  ou si eval($expr_1$) = { } & eval($expr_2$) = { }
- FALSE
  si eval($expr_1$) ≠ error & eval($expr_1$) ≠ { }
  & eval($expr_2$) ≠ error & eval($expr_2$) ≠ { }
  & ∀ x ∈ eval($expr_1$)
  :
  (inh-ins($expr_2$) = x
  & inh-Table($expr_2$) = instX
  & (x, instX) ∈ Table($expr_1$)
)
  ⟹∀y ∈ eval($expr_2$) : (∃el ∈ x : el ∉ y)ou(∃el ∈ y : el∉x)
- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins($expr_1$) = inh-ins(expr)
inh-Table($expr_1$) = inh-Table(expr)
∀ (x,instX) ∈ Table($expr_1$) :
  inh-ins($expr_2$) = x
  inh-Table($expr_2$) = instX
Table(expr) = selon les cas :

- $\bigcup\left[\bigcup\left[\{(A, B)\}\right]\right|_{\forall y \in eval(expr2)}\left.\right]\Big|_{\forall x \in eval(expr1)}$
  ∀ (A, B) où
  A = inh-ins(expr), B = instY
  (x, instX) ∈ Table($expr_1$)
  inh-ins($expr_2$) = x, inh-Table($expr_2$) = instX
  (y, instY) ∈ Table($expr_2$)
  (∀ el ∈ x : el ∈ y) & (∀ el ∈ y : el ∈ x)
  si eval(expr) = TRUE
- { }
  si eval(expr) = FALSE
- empty
  sinon

Attribut denote

denote($expr_1$) = denote(expr)

denote($expr_2$) = denote(expr)

Attribut insForEval

insForEval($expr_1$) = insForEval(expr)

insForEval($expr_2$) = insForEval(expr)

7. Opérateurs définissant des fonctions agrégées

1. expr = AVG(expr1)

Attribut eval

eval(expr) = selon les cas :

- { av, ∀ av = average(X), X ∈ eval($expr_1$) }
  si eval($expr_1$) ≠ error & eval($expr_1$) ≠ { }
  & X ∈ eval($expr_1$)
  & X ≠ { }
  & inh-Table(expr) ≠ empty
- error
  sinon

Note : $\text{average}(X) = \left(\sum [\, x_i \,] \mid_{(x_i \in X)} \right) / \#(X)$

Attributs inh-Table, inh-ins et Table

inh-ins($expr_1$) = inh-ins(expr)

inh-Table($expr_1$) = inh-Table(expr)

Table(expr) = selon les cas :

- { (val, instval),
  ∀ val ∈ eval(expr)
  & val = average(X)
  & (X, instval) ∈ Table($expr_1$) }
  si eval(expr) ≠ error
- empty
  sinon

Attribut denote

denote($expr_1$) = denote(expr)

Attribut insForEval

insForEval($expr_1$) = insForEval(expr)

2. expr = MIN(expr1)

Attribut eval

eval(expr) = selon les cas :

- { min, ∀ min :
  ((min <= elem,∀ elem ∈ X)& min ∈ X), ∀X ∈ eval($expr_1$)
  }
  si eval($expr_1$) ≠ error & eval($expr_1$) ≠ { }
  & X ∈ eval($expr_1$)
  & X ≠ { }
  & inh-Table(expr) ≠ empty
- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins($expr_1$) = inh-ins(expr)

inh-Table($expr_1$) = inh-Table(expr)

Table(expr) = selon les cas :

- { (val, instval),
  ∀ val ∈ eval(expr)
  & (val <= elem, ∀ elem ∈ X) & val ∈ X
  & (X, instval) ∈ Table($expr_1$)
  }
  si eval(expr) ≠ error
- empty
  sinon

Attribut denote

denote($expr_1$) = denote(expr)

Attribut insForEval

insForEval($expr_1$) = insForEval(expr)

3. expr = MAX(expr1)

Attribut eval

eval(expr) = selon les cas :

- { max, ∀ max :
  ((max >= elem,∀elem ∈ X)& max ∈ X), ∀X ∈ eval($expr_1$)
  }
  si eval($expr_1$) ≠ error & eval($expr_1$) ≠ { }
  & X ∈ eval($expr_1$)
  & X ≠ { }
  & inh-Table(expr) ≠ empty
- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins($expr_1$) = inh-ins(expr)

inh-Table($expr_1$) = inh-Table(expr)

Table(expr) = selon les cas :

- { (val, instval),
  ∀ val ∈ eval(expr)
  & (val >= elem, ∀ elem ∈ X) & val ∈ X
  & (X, instval) ∈ Table($expr_1$)
  }
  si eval(expr) ≠ error
- empty
  sinon

Attribut denote

denote($expr_1$) = denote(expr)

Attribut insForEval

insForEval($expr_1$) = insForEval(expr)

4. expr = SUM(expr1)

Attribut eval

eval(expr) = selon les cas :

- { sum, ∀ sum :

  $(sum = \Sigma [x_i] \mid_{(xi \in X)}), \forall X \in eval(expr_1)$

  }

  si $eval(expr_1) \neq error$ & $eval(expr_1) \neq \{ \}$

  & $X \in eval(expr_1)$ & $X \neq \{ \}$

  & inh-Table(expr) ≠ empty
- error

  sinon

Attributs inh-Table, inh-ins et Table

inh-ins($expr_1$) = inh-ins(expr)

inh-Table($expr_1$) = inh-Table(expr)

Table(expr) = selon les cas :

- { (val, instval), ∀ val ∈ eval(expr)

  & $(val = \Sigma [x_i] \mid_{(xi \in X)})$

  & (X, instval) ∈ Table($expr_1$)

  }

  si eval(expr) ≠ error
- empty

  sinon

Attribut denote

denote($expr_1$) = denote(expr)

Attribut insForEval

insForEval($expr_1$) = insForEval(expr)

5. expr = COUNT(expr1)

Attribut eval

eval(expr) = selon les cas :

- { count, ∀ count :
  (count = #(X), ∀X ∈ eval($expr_1$)
  }
  si eval($expr_1$) ≠ error & eval($expr_1$) ≠ { }
  & X ∈ eval($expr_1$)
  & X ≠ { }
  & inh-Table(expr) ≠ empty
- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins($expr_1$) = inh-ins(expr)

inh-Table($expr_1$) = inh-Table(expr)

Table(expr) = selon les cas :

- { (val, instval),
  ∀ val ∈ eval(expr)
  & val = #(X),
  & (X, instval) ∈ Table($expr_1$)
  }
  si eval(expr) ≠ error
- empty
  sinon

Attribut denote

denote($expr_1$) = denote(expr)

Attribut insForEval

insForEval($expr_1$) = insForEval(expr)

8. Opérateurs arithmétiques

1. expr = PLUS(expr1, expr2)

Attribut eval

eval(expr) = selon les cas :

- { x + y, ∀ x ∈ eval($expr_1$), y ∈ eval($expr_2$) }
  si eval($expr_1$) ≠ error & eval($expr_1$) ≠ { }
  & eval($expr_2$) ≠ error & eval($expr_2$) ≠ { }
  & x ∈ eval($expr_1$)
  & inh-ins($expr_2$) = x
  & inh-Table($expr_2$) = instX
  & (x,instX) ∈ Table($expr_1$)
  & y ∈ eval($expr_2$)
- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins($expr_1$) = inh-ins(expr)
inh-Table($expr_1$) = inh-Table(expr)
∀ (x,inst) ∈ Table($expr_1$) :
  inh-ins($expr_2$) = x
  inh-Table($expr_2$) = inst
Table(expr) = selon les cas :

- { (val, instval),
    ∀ val ∈ eval(expr)
    & val = a + b
    & (a,insta) ∈ Table($expr_1$)
    & inh-ins($expr_2$) = a
    & inh-Table($expr_2$) = insta
    & (b,instval) ∈ Table($expr_2$)
  }
  si eval(expr) ≠ error
- empty
  sinon

Attribut denote

denote($expr_1$) = denote(expr)
denote($expr_2$) = denote(expr)

Attribut insForEval

insForEval($expr_1$) = insForEval(expr)
insForEval($expr_2$) = insForEval(expr)

2. expr = MINUS (expr1, expr2)

Attribut eval

eval(expr) = selon les cas :

- { x - y, ∀ x ∈ eval($expr_1$), y ∈ eval($expr_2$) }
  si eval($expr_1$) ≠ error & eval($expr_1$) ≠ { }
  & eval($expr_2$) ≠ error & eval($expr_2$) ≠ { }
  & x ∈ eval($expr_1$)
  & inh-ins($expr_2$) = x
  & inh-Table($expr_2$) = instX
  & (x,instX) ∈ Table($expr_1$)
  & y ∈ eval($expr_2$)
- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins($expr_1$) = inh-ins(expr)
inh-Table($expr_1$) = inh-Table(expr)
∀ (x,inst) ∈ Table($expr_1$) :
  inh-ins($expr_2$) = x
  inh-Table($expr_2$) = inst
Table(expr) = selon les cas :

- { (val, instval),
    ∀ val ∈ eval(expr)
    & val = a - b
    & (a,insta) ∈ Table($expr_1$)
    & inh-ins($expr_2$) = a
    & inh-Table($expr_2$) = insta
    & (b,instval) ∈ Table($expr_2$)
  }
  si eval(expr) ≠ error
- empty
  sinon

Attribut denote

denote($expr_1$) = denote(expr)
denote($expr_2$) = denote(expr)

Attribut insForEval

insForEval($expr_1$) = insForEval(expr)
insForEval($expr_2$) = insForEval(expr)

3. expr = TIMES(expr1, expr2)

Attribut eval

eval(expr) = selon les cas :

- { x * y, ∀ x ∈ eval($expr_1$), y ∈ eval($expr_2$) }
  si eval($expr_1$) ≠ error & eval($expr_1$) ≠ { }
  & eval($expr_2$) ≠ error & eval($expr_2$) ≠ { }
  & x ∈ eval($expr_1$)
  & inh-ins($expr_2$) = x
  & inh-Table($expr_2$) = instX
  & (x,instX) ∈ Table($expr_1$)
  & y ∈ eval($expr_2$)
- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins($expr_1$) = inh-ins(expr)
inh-Table($expr_1$) = inh-Table(expr)
∀ (x,inst) ∈ Table($expr_1$) :
  inh-ins($expr_2$) = x
  inh-Table($expr_2$) = inst
Table(expr) = selon les cas :

- { (val, instval),
    ∀ val ∈ eval(expr)
    & val = a * b
    & (a,insta) ∈ Table($expr_1$)
    & inh-ins($expr_2$) = a
    & inh-Table($expr_2$) = insta
    & (b,instval) ∈ Table($expr_2$)
  }
  si eval(expr) ≠ error
- empty
  sinon

Attribut denote

denote($expr_1$) = denote(expr)
denote($expr_2$) = denote(expr)

Attribut insForEval

insForEval($expr_1$) = insForEval(expr)
insForEval($expr_2$) = insForEval(expr)

4. expr = DIV(expr1, expr2)

Attribut eval

eval(expr) = selon les cas :

- { x / y, ∀ x ∈ eval($expr_1$), y ∈ eval($expr_2$) }
  si eval($expr_1$) ≠ error & eval($expr_1$) ≠ { }
  & eval($expr_2$) ≠ error & eval($expr_2$) ≠ { }
  & x ∈ eval($expr_1$)
  & inh-ins($expr_2$) = x
  & inh-Table($expr_2$) = instX
  & (x,instX) ∈ Table($expr_1$)
  & y ∈ eval($expr_2$)
- error
  sinon

Attributs inh-Table, inh-ins et Table

inh-ins($expr_1$) = inh-ins(expr)
inh-Table($expr_1$) = inh-Table(expr)
∀ (x,inst) ∈ Table($expr_1$) :
  inh-ins($expr_2$) = x
  inh-Table($expr_2$) = inst
Table(expr) = selon les cas :

- { (val, instval),
    ∀ val ∈ eval(expr)
    & val = a / b
    & (a,insta) ∈ Table($expr_1$)
    & inh-ins($expr_2$) = a
    & inh-Table($expr_2$) = insta
    & (b,instval) ∈ Table($expr_2$)
  }
  si eval(expr) ≠ error
- empty
  sinon

Attribut denote

denote($expr_1$) = denote(expr)
denote($expr_2$) = denote(expr)

Attribut insForEval

insForEval($expr_1$) = insForEval(expr)
insForEval($expr_2$) = insForEval(expr)

3.2. Sémantique d'un slot

Soit une définition de classe Cl contenant un slot S.

Le slot S slot possède une facette (def, PCLexpression), pour laquelle on désire définir la sémantique.

expr = PCLexpression

Attribut eval
eval(expr) = eval(PCLexpression)

Attributs inh-Table, inh-ins et Table
inh-ins(PCLexpression) = no-value
inh-Table(PCLexpression) = { }
Table(expr) = Table(PCLexpression)

Attribut denote
denote(PCLexpression) = val

Attribut insForEval
insForEval(PCLexpression) = Cl

On peut également vouloir évaluer le slot S.

expr = slotname

Attribut eval
eval(expr) = eval(slotname)

Attributs inh-Table, inh-ins et Table
inh-ins(slotname) = no-value
inh-Table(slotname) = { }
Table(expr) = Table(slotname)

Attribut denote
denote(slotname) = val

Attribut insForEval
insForEval(slotname) = Cl

3.3. Sémantique d'une requête

Soit une requête, pour laquelle on désire définir la sémantique.

expr = PCLquery

Attribut eval
eval(expr) = eval(PCLquery)

Attributs inh-Table, inh-ins et Table
inh-ins(PCLexpression) = no-value
inh-Table(PCLquery) = { }
Table(expr) = Table(PCLquery)

Attribut denote
denote(PCLquery) = val

Attribut insForEval
insForEval(PCLquery) = no-value

Note (1) :

On peut désirer spécifier une instance (notée ins) pour laquelle on désire évaluer l'expression expr.

Dans ce cas, cette instance est dénotée par THIS dans l'expression PCL. La sémantique de l'expression expr est la sémantique de l'expression formée en remplaçant THIS par une variable (notée var) dans expr, et en attribuant à cette variable la valeur ins.

Attributs inh-Table, inh-ins et Table
inh-ins(PCLexpression) = ins
inh-Table(PCLquery) = { ‹var, Cl, ins› }
Table(expr) = Table(PCLquery)

4. Contrainte sur le modèle conceptuel

Après avoir défini la sémantique de chaque expression de la syntaxe abstraite, on peut exprimer la contrainte suivante sur le modèle conceptuel :

"L'évaluation de chaque slot

=

la valeur de ce slot fixée au niveau TROIS de la sémantique"

Choix d'une syntaxe concrète

Annexe 8

1. Introduction

Ce document presente la syntaxe concrete choisie pour langage PCL. Les deux premiers points décrivent les conventions utilisées, ainsi que la manière dont la syntaxe concrète sera présentée. Le dernier point donne la description de la syntaxe du langage.

2. Conventions d'écriture

Tous les mots définis dans le langage PCL seront en caractères gras (opérateurs, mots définis, ...), tandis que les mots en italiques désignent des expressions PCL plus ou moins complexes (noms de slot, noms de classe, expressions très générales, ...).

3. Présentation de la syntaxe

La syntaxe du langage PCL est présentée de la manière suivante :

1. Expressions primitives :

Pour chaque type d'expression primitive, on précisera les règles de représentation, ainsi qu'un exemple.

2. Opérateurs :

Les opérateurs sont classés par famille.

Chaque opérateur est défini suivant le plan ci-dessous.

Syntaxe :

La syntaxe est donnée en respectant les conventions d'écriture ci-dessus.

Description :

La description de l'opérateur comprend :

- l'**arité** de l'opérateur.(nombre d'opérandes)
- **position** de l'opérateur (d'arité 1 ou 2 uniquement) par rapport à ces opérandes :

  infixe (opérateur placé entre les opérandes),
  postfixe (opérandes précédant l'opérateur),
  préfixe (opérateur précédant les opérandes).

- l'**associativité** (des opérateurs d'arité 1 ou 2 uniquement) :

  pour chaque opérande, la relation :

  <= prior(opérateur),

  si la priorité maximale de l'opérande doit être inférieure ou égale à la priorité de l'opérateur

  < prior(opérateur),

  si la priorité maximale de l'opérande doit être strictement inférieure à la priorité de l'opérateur

- le niveau de **priorité** : un entier compris entre 1 et 10

- pour chaque **argument**, une indication sur sa forme :

  expression PCL générale,
  expression restreinte à un nom de slot,
  expression restreinte à un nom de classe,
  variable,
  tuple.

- le type de l'expression.

- un énoncé bref de la sémantique associée à l'opérateur.

Cette description est basée sur la méthode de définition des opérateurs en Prolog. Elle a cependant été étendue aux opérateurs d'arité 3.

Resctrictions :

Dans certains cas, certaines restrictions sont imposées sur l'un des arguments.

Concordance de type :

Les règles de concordance de types sont données sous la forme d'une table ayant les caractéristiques suivantes :

- les n premières colonnes désignent le type des n arguments de l'opérateur,
- la colonne notée "comp" indique si l'expression est correcte,
- la colonne notée "result type" donne le type de l'expression.

4. Syntaxe

4.1. Expressions de base

1. Constantes :

Les constantes PCL peuvent être de trois types :

- les constantes numériques sont représentées par un entier.

  ex : 12, 46, 3579, ...

- les constantes réelles sont représentées par un réel.

  ex : 12.45, 46.869, 3579.8556, ...

- les constantes de type *string* sont représentées par une chaîne de caractères entre guillemets.

  ex : "Durant", "Paris", "vert", ...

2. Classes :

Les noms de classes PCL doivent obligatoirement commencer par une lettre majuscule.

3. Slots :

Les noms de slots PCL doivent obligatoirement commencer par une lettre minuscule.

4. Variables :

Aucune restriction n'est imposée sur les noms de variables, celles-ci étant obligatoirement précédées de l'opérateur "?".

ex : ? X, ? x, ? somme, ...

5. Ensembles explicites de valeurs :

Un ensemble explicite de valeurs s'exprime sous la forme suivante :

- [x_1, x_2, ... ,x_n] , où chaque x_i est soit une constante, soit un nom de slot, et tel que tous les x_i soient de même type.

  ex : [4, 5, 8, 4556, 7889] , ["vert", "rouge", "jaune"] , ...

6. Tuples :

Un tuple s'exprime sous la forme suivante :

- [$term_1$, $term_2$, ... ,$term_n$] , où chaque $term_i$ est une expression PCL.

4.2. Opérateurs

4.2.1. Opérateur définissant un lien entre un slot et une classe

1. L'opérateur # :

Syntaxe :

expression # *slotname*

Description :

arité : 2

position : infixe

associativité :

1. < prior(opérateur)
2. <= prior(opérateur)

priorité : 2

arguments :

1. un nom de classe, une variable désignant une classe, une expression de la forme expr1 # expr2
2. un nom de slot

type : le type du nom de slot

sémantique :

L'expression *expression* # *slotname* signifie que le slot de nom *slotname* est défini pour la classe référenciée par l'expression *expression* .

Resctrictions :

Dans le cas où le premier argument est une variable, celle-ci doit déjà avoir été définie et désigner une classe pour laquelle le slot de nom *slotname* est défini.

Concordance de type :

expr1	expr2	comp	result type
class1	set(class2),class2	correct	class1
integer	set(class2),class2	correct	integer
real	set(class2),class2	correct	real
string	set(class2),class2	correct	string
set(class1)	set(class2),class2	correct	set(class1)
set(integer)	set(class2),class2	correct	set(integer)
set(real)	set(class2),class2	correct	set(real)
set(string)	set(class2),class2	correct	set(string)
others	others	not correct	undetermined

où

- class1, class2 sont des classes dites "non-basic",
- class1 peut être différente de class2

4.2.2. Opérateur définissant une variable

1. L'opérateur ? :

Syntaxe :

? *variable*

Description :

arité : 1

position : préfixe

associativité :

1. < prior(opérateur)

priorité : 1

argument :

1. une variable

type : le type de la variable

sémantique :

L'expression **?** *variable* signifie que le nom *variable* désigne une variable.

4.2.3. Opérateur définissant un ensemble de valeurs

1. SETOF :

Syntaxe :

SETOF *PCLexpr*

Description :

arité : 1

position : préfixe

associativité :

1. <= prior(opérateur)

priorité : 5

argument :

1. un nom de classe, une variable, un tuple

type : set(X), où X est le type le l'expression *PCLexpr*

sémantique :

L'expression **SETOF** *PCLexpr* désigne un ensemble pouvant être vide d'objets définis par l'expression *PCLexpr* .

Resctrictions :

Dans le cas où l'argument est une variable, celle-ci doit déjà avoir été définie et désigner une classe.

Concordance de type :

expr1	comp	result type
tuple	correct	set(tuple)
class1	correct	set(class1)
integer	correct	set(integer)
real	correct	set(real)
string	correct	set(string)
others	not correct	undetermined

où

- classl est une classe dite "non-basic".

4.2.4. Opérateur définissant une condition sur un objet

1. L'opérateur WHERE :

Syntaxe :

expr **WHERE** *booleanExpr*

Description :

arité : 2

position : infixe

associativité :

1. < prior(opérateur)
2. <= prior(opérateur)

priorité : 4

arguments :

1. un nom de classe, une variable
2. une expression booléenne

type : le type de l'expression *expr*

sémantique :

L'expression *expr* **WHERE** *booleanExpr* désigne un objet (défini par l'expression *expr*) satisfaisant la condition booléenne définie par *booleanExpr*.

Notons :

Si plusieurs objets satisfont cette expresssion, les différentes valeurs pour l'expression peuvent être obtenues par backtracking.

Concordance de type :

expr1	expr2	comp	result type
class1 others	boolean others	correct not correct	class1 undetermined

où

- class1 est une classe dite "non-basic".

4.2.5. Opérateurs logiques

1. L'opérateur NOT :

Syntaxe :

NOT *booleanExpr*

Description :

arité : 1

position : préfixe

associativité :

1. <= prior(opérateur)

priorité : 8

argument :

1. une expression booléenne

type : boolean

sémantique :

L'expression **NOT** *booleanExpr* prend la valeur :

- TRUE si l'expression booléenne *booleanExpr* prend la valeur FALSE,
- FALSE si l'expression booléenne *booleanExpr* prend la valeur TRUE.

Concordance de type :

expr1	comp	result type
boolean others	correct not correct	boolean undetermined

2. L'opérateur AND :

Syntaxe :

booleanExpr1 **AND** *booleanExpr2*

Description :

arité : 2

position : infixe

associativité :

1. < prior(opérateur)
2. <= prior(opérateur)

priorité : 9

arguments :

1. une expression booléenne
2. une expression booléenne

type : boolean

sémantique :

L'expression *booleanExpr1* **AND** *booleanExpr2* prend la valeur :

- TRUE si les expressions booléennes *booleanExpr1* et *boleanExpr2* prennent la valeur TRUE,
- FALSE si l'une des expressions booléennes *booleanExpr1* ou *booleanExpr2* prend la valeur FALSE.

Concordance de type :

expr1	expr2	comp	result type
boolean others	boolean others	correct not correct	boolean undetermined

3. L'opérateur OR :

Syntaxe :

booleanExpr1 **OR** *booleanExpr2*

Description :

arité : 2

position : infixe

associativité :

1. < prior(opérateur)
2. <= prior(opérateur)

priorité : 10

arguments :

1. une expression booléenne
2. une expression booléenne

type : boolean

sémantique :

L'expression *booleanExpr1* **OR** *booleanExpr2* prend la valeur :

- TRUE si l'une des expressions booléennes *booleanExpr1* ou *booleanExpr2* prend la valeur TRUE,
- FALSE si les expressions booléennes *booleanExpr1* et *boleanExpr2* prennent la valeur FALSE.

Concordance de type :

expr1	expr2	comp	result type
boolean others	boolean others	correct not correct	boolean undetermined

4. L'opérateur EXIST ... MEMBER ... WITH :

Syntaxe :

EXIST *expr1* **MEMBER** *expr2* **WITH** *booleanExpr3*

Description :

arité : 3

priorité : 5

arguments :

1. une variable non déjà définie
2. une expression PCL
3. une condition booléenne

type : boolean

sémantique :

L'expression

EXIST *expr1* **MEMBER** *expr2* **WITH** *booleanExpr3*

prend la valeur :

- TRUE s'il existe une valeur "X" de l'ensemble défini par *expr2* pour laquelle l'évaluation de l'expression *booleanExpr3* donne la valeur TRUE.

  Dans ce cas, la variable *expr1* prend cette valeur "X".

- FALSE si pour toutes les valeurs de l'ensemble défini par *expr2* l'évaluation de l'expression *booleanExpr3* donne la valeur FALSE.

  Dans ce cas, la variable *expr1* ne prend aucune valeur.

Resctrictions :

Dans le cas où le deuxième argument est une variable, celle-ci doit déjà avoir été définie et désigner un ensemble de valeurs.

Concordance de type :

expr1	expr2	expr3	comp	result type
tuple	set(tuple)	boolean	correct	boolean
class1	set(class1)	boolean	correct	boolean
integer	set(integer)	boolean	correct	boolean
real	set(real)	boolean	correct	boolean
string	set(string)	boolean	correct	boolean
others	others	others	not correct	undetermined

où

- class1 est une classe dite "non-basic".

5. L'opérateur EXIST ... INCLUDED ... WITH :

Syntaxe :

EXIST *expr1* **INCLUDED** *expr2* **WITH** *booleanExpr3*

Description :

arité : 3

priorité : 5

arguments :

1. une variable non déjà définie
2. une expression PCL
3. une condition booléenne

type : boolean

sémantique :

L'expression

EXIST *expr1* **INCLUDED** *expr2* **WITH** *booleanExpr3*

prend la valeur :

- TRUE s'il existe un sous-ensemble "X" de l'ensemble défini par *expr2* pour lequelle l'évaluation de l'expression *booleanExpr3* donne la valeur TRUE.

  Dans ce cas, la variable *expr1* prend cette valeur "X".

- FALSE si pour tout sous-ensemble de l'ensemble défini par *expr2*, l'évaluation de l'expression *booleanExpr3* donne la valeur FALSE.

  Dans ce cas, la variable *expr1* ne prend aucune valeur.

Resctrictions :

Dans le cas où le deuxième argument est une variable, celle-ci doit déjà avoir été définie et désigner un ensemble de valeurs.

Concordance de type :

expr1	expr2	expr3	comp	result type
set(tuple)	set(tuple)	boolean	correct	boolean
set(class1)	set(class1)	boolean	correct	boolean
set(integer)	set(integer)	boolean	correct	boolean
set(real)	set(real)	boolean	correct	boolean
set(string)	set(string)	boolean	correct	boolean
others	others	others	not correct	undetermined

où

- class1 est une classe dite "non-basic".

6. L'opérateur FORALL ... MEMBER ... WITH :

Syntaxe :

FORALL *expr1* **MEMBER** *expr2* **WITH** *booleanExpr3*

Description :

arité : 3

priorité : 5

arguments :

1. une variable non déjà définie
2. une expression PCL
3. une condition booléenne

type : boolean

sémantique :

L'expression

FORALL *expr1* **MEMBER** *expr2* **WITH** *booleanExpr3*

prend la valeur :

- TRUE si pour toutes les valeurs de l'ensemble défini par *expr2*, l'évaluation de l'expression *booleanExpr3* donne la valeur TRUE.

  Dans ce cas, la variable *expr1* ne prend aucune valeur.

- FALSE s'il existe une valeur "X" de l'ensemble défini par *expr2* pour laquelle l'évaluation de l'expression *booleanExpr3* ne donne pas la valeur TRUE.

  Dans ce cas, la variable *expr1* ne prend aucune valeur.

Resctrictions :

Dans le cas où le deuxième argument est une variable, celle-ci doit déjà avoir été définie et désigner un ensemble de valeurs.

Concordance de type :

expr1	expr2	expr3	comp	result type
tuple	set(tuple)	boolean	correct	boolean
class1	set(class1)	boolean	correct	boolean
integer	set(integer)	boolean	correct	boolean
real	set(real)	boolean	correct	boolean
string	set(string)	boolean	correct	boolean
others	others	others	not correct	undetermined

où

- class1 est une classe dite "non-basic".

7. L'opérateur FORALL ... INCLUDED ... WITH :

Syntaxe :

FORALL *expr1* **INCLUDED** *expr2* **WITH** *booleanExpr3*

Description :

arité : 3

priorité : 5

arguments :

1. une variable non déjà définie
2. une expression PCL
3. une condition booléenne

type : boolean

sémantique :

L'expression

FORALL *expr1* **INCLUDED** *expr2* **WITH** *booleanExpr3*

prend la valeur :

- TRUE si pour tout sous-ensemble de l'ensemble défini par *expr2*, l'évaluation de l'expression *booleanExpr3* donne la valeur TRUE.

  Dans ce cas, la variable *expr1* ne prend aucune valeur.

- FALSE s'il existe un sous-ensemble "X" de l'ensemble défini par *expr2* pour lequelle l'évaluation de l'expression *booleanExpr3* ne donne pas la valeur TRUE.

  Dans ce cas, la variable *expr1* ne prend aucune valeur.

Resctrictions :

Dans le cas où le deuxième argument est une variable, celle-ci doit déjà avoir été définie et désigner un ensemble de valeurs.

Concordance de type :

expr1	expr2	expr3	comp	result type
set(tuple)	set(tuple)	boolean	correct	boolean
set(class1)	set(class1)	boolean	correct	boolean
set(integer)	set(integer)	boolean	correct	boolean
set(real)	set(real)	boolean	correct	boolean
set(string)	set(string)	boolean	correct	boolean
others	others	others	not correct	undetermined

où

- class1 est une classe dite "non-basic".

4.2.6. Opérateurs de comparaison

1. L'opérateur EQ :

Syntaxe :

expr1 **EQ** *expr2*

Description :

arité : 2

position : infixe

associativité :

1. < prior(opérateur)
2. < prior(opérateur)

priorité : 3

arguments :

1. une expression PCL
2. une expression PCL

type : boolean

sémantique :

L'expression *expr1* **EQ** *expr2* prend la valeur :

- TRUE si la valeur de l'expression *expr1* est égale à la valeur de l'expression *expr2* .
- FALSE si la valeur de l'expression *expr1* n'est pas égale à la valeur de l'expression *expr2* .

Resctrictions :

Dans le cas où le deuxième argument est une variable, celle-ci doit déjà avoir été définie et désigner une valeur simple.

On notera également que si le premier argument est une variable non déjà apparue, celle-ci est alors définie et prend la même valeur que le deuxième argument.

Concordance de type :

expr1	expr2	comp	result type
tuple	tuple	correct	boolean
class1	class2	correct	boolean
integer	integer	correct	boolean
real	real	correct	boolean
string	string	correct	boolean
others	others	not correct	undetermined

où

- class1, class2 sont des classes dites "non-basic".
- class1 peut être différente de class2

2. L'opérateur NE :

Syntaxe :

expr1 **NE** *expr2*

Description :

arité : 2

position : infixe

associativité :

1. < prior(opérateur)
2. < prior(opérateur)

priorité : 3

arguments :

1. une expression PCL
2. une expression PCL

type : boolean

sémantique :

L'expression *expr1* **NE** *expr2* prend la valeur :

- TRUE si la valeur de l'expression *expr1* n'est pas égale à la valeur de l'expression *expr2*.
- FALSE si la valeur de l'expression *expr1* est égale à la valeur de l'expression *expr2*.

Resctrictions :

Dans le cas ou l'un des arguments est une variable, celle-ci doit déjà avoir été définie et désigner une valeur simple.

Concordance de type :

expr1	expr2	comp	result type
tuple	tuple	correct	boolean
class1	class2	correct	boolean
integer	integer	correct	boolean
real	real	correct	boolean
string	string	correct	boolean
others	others	not correct	undetermined

où

- class1, class2 sont des classes dites "non-basic".
- class1 peut être différente de class2

3. L'opérateur GT :

Syntaxe :

expr1 **GT** *expr2*

Description :

arité : 2

position : infixe

associativité :

1. < prior(opérateur)
2. < prior(opérateur)

priorité : 3

arguments :

1. une expression PCL
2. une expression PCL

type : boolean

sémantique :

L'expression *expr1* **GT** *expr2* prend la valeur :

- TRUE si la valeur de l'expression *expr1* est strictement plus grande que la valeur de l'expression *expr2* .
- FALSE si la valeur de l'expression *expr1* est plus petite ou égale à la valeur de l'expression *expr2* .

Resctrictions :

Dans le cas où l'un des arguments est une variable, celle-ci doit déjà avoir été définie et désigner une valeur simple.

Concordance de type :

expr1	expr2	comp	result type
integer	integer	correct	boolean
integer	real	correct	boolean
real	integer	correct	boolean
real	real	correct	boolean
string	string	correct	boolean
others	others	not correct	undetermined

4. L'opérateur GE :

Syntaxe :

expr1 **GE** *expr2*

Description :

arité : 2

position : infixe

associativité :

1. < prior(opérateur)
2. < prior(opérateur)

priorité : 3

arguments :

1. une expression PCL
2. une expression PCL

type : boolean

sémantique :

L'expression *expr1* **GE** *expr2* prend la valeur :

- TRUE si la valeur de l'expression *expr1* est plus grande ou égale à la valeur de l'expression *expr2* .
- FALSE si la valeur de l'expression *expr1* est strictement plus petite que la valeur de l'expression *expr2* .

Resctrictions :

Dans le cas où l'un des arguments est une variable, celle-ci doit déjà avoir été définie et désigner une valeur simple.

Concordance de type :

expr1	expr2	comp	result type
integer	integer	correct	boolean
integer	real	correct	boolean
real	integer	correct	boolean
real	real	correct	boolean
string	string	correct	boolean
others	others	not correct	undetermined

5. L'opérateur ST :

Syntaxe :

expr1 **ST** *expr2*

Description :

arité : 2

position : infixe

associativité :

1. < prior(opérateur)
2. < prior(opérateur)

priorité : 3

arguments :

1. une expression PCL
2. une expression PCL

type : boolean

sémantique :

L'expression *expr1* **ST** *expr2* prend la valeur :

- TRUE si la valeur de l'expression *expr1* est strictement plus petite que la valeur de l'expression *expr2* .
- FALSE si la valeur de l'expression *expr1* est plus grande ou égale à la valeur de l'expression *expr2* .

Resctrictions :

Dans le cas où l'un des arguments est une variable, celle-ci doit déjà avoir été définie et désigner une valeur simple.

Concordance de type :

expr1	expr2	comp	result type
integer	integer	correct	boolean
integer	real	correct	boolean
real	integer	correct	boolean
real	real	correct	boolean
string	string	correct	boolean
others	others	not correct	undetermined

6. L'opérateur SE :

Syntaxe :

expr1 **SE** *expr2*

Description :

arité : 2

position : infixe

associativité :

1. < prior(opérateur)
2. < prior(opérateur)

priorité : 3

arguments :

1. une expression PCL
2. une expression PCL

type : boolean

sémantique :

L'expression *expr1* **SE** *expr2* prend la valeur :

- TRUE si la valeur de l'expression *expr1* est plus petite ou égale à la valeur de l'expression *expr2* .
- FALSE si la valeur de l'expression *expr1* strictement plus grande que la valeur de l'expression *expr2* .

Resctrictions :

Dans le cas où l'un des arguments est une variable, celle-ci doit déjà avoir été définie et désigner une valeur simple.

Concordance de type :

expr1	expr2	comp	result type
integer	integer	correct	boolean
integer	real	correct	boolean
real	integer	correct	boolean
real	real	correct	boolean
string	string	correct	boolean
others	others	not correct	undetermined

4.2.7. Opérateurs ensemblistes

1. L'opérateur MEMBER :

Syntaxe :

expr1 **MEMBER** *expr2*

Description :

arité : 2

position : infixe

associativité :

1. < prior(opérateur)
2. < prior(opérateur)

priorité : 3

arguments :

1. une expression PCL
2. une expression PCL

type : boolean

sémantique :

L'expression *expr1* **MEMBER** *expr2* prend la valeur :

- TRUE si la valeur de l'expression *expr1* est une des valeurs de l'ensemble défini par l'expression *expr2* .
- FALSE si la valeur de l'expression *expr1* n'est pas une des valeurs de l'ensemble défini par l'expression *expr2*.

Resctrictions :

Dans le cas où l'un des arguments est une variable, celle-ci doit déjà avoir été définie et posséder une valeur.

Concordance de type :

expr1	expr2	comp	result type
tuple	set(tuple)	correct	boolean
class1	set(class2)	correct	boolean
integer	set(integer)	correct	boolean
real	set(real)	correct	boolean
string	set(string)	correct	boolean
others	others	not correct	undetermined

où

- class1, class2 sont des classes dites "non-basic".

- classl peut être :
    - la même que class2
    - une sous-classe de class2

2. L'opérateur INCLUDED :

Syntaxe :

expr1 **INCLUDED** *expr2*

Description :

arité : 2

position : infixe

associativité :

1. < prior(opérateur)
2. < prior(opérateur)

priorité : 3

arguments :

1. une expression PCL
2. une expression PCL

type : boolean

sémantique :

L'expression *expr1* **INCLUDED** *expr2* prend la valeur :

- TRUE si l'ensemble de valeurs défini par l'expression *expr1* est un sous-ensemble de l'ensemble défini par l'expression *expr2*.
- FALSE si l'ensemble de valeurs défini par l'expression *expr1* n'est pas un sous-ensemble de l'ensemble défini par l'expression *expr2*.

Resctrictions :

Dans le cas où le deuxième argument est une variable, celle-ci doit déjà avoir été définie et posséder une valeur.

Concordance de type :

expr1	expr2	comp	result type
set(tuple)	set(tuple)	correct	boolean
set(class1)	set(class2)	correct	boolean
set(integer)	set(integer)	correct	boolean
set(real)	set(real)	correct	boolean
set(string)	set(string)	correct	boolean
others	others	not correct	undetermined

où

- class1, class2 sont des classes dites "non-basic".
- class1 peut être :
    - la même que class2
    - une sous-classe de class2

3. L'opérateur ISIN :

Syntaxe :

expr1 **ISIN** *expr2*

Description :

arité : 2

position : infixe

associativité :

1. < prior(opérateur)
2. < prior(opérateur)

priorité : 3

arguments :

1. une expression PCL
2. une expression PCL

type : boolean

sémantique :

L'expression *expr1* **ISIN** *expr2* prend la valeur :

- TRUE si la valeur de l'expression *expr1* est une valeur possible pour l'expression *expr2* , au sens de l'appartenance à une définition de type.
- FALSE si la valeur de l'expression *expr1* n'est pas une valeur possible pour l'expression *expr2* , au sens de l'appartenance à une définition de type.

Resctrictions :

- Cas où le deuxième argument n'est pas une variable :

Dans le cas où le premier argument est une variable, celle-ci doit déjà avoir été définie et posséder une valeur. Il faut alors également restreindre les règles de concordance de types aux seuls cas où le deuxième argument est de type class2 ou set(class2).

- Cas où le deuxième argument est une variable :

La variable définissant le deuxième argument doit déjà avoir été définie et posséder une valeur. Il faut alors restreindre les règles de concordance de types aux seuls cas où le deuxième argument est de type class2 ou set(class2).

Si le premier argument est également une variable, celle-ci doit aussi avoir été définie et posséder une valeur.

Concordance de type :

expr1	expr2	comp	result type
class1	class2	correct	boolean
integer	integer	correct	boolean
real	real	correct	boolean
string	string	correct	boolean
set(class1)	set(class2)	correct	boolean
set(integer)	set(integer)	correct	boolean
set(real)	set(real)	correct	boolean
set(string)	set(string)	correct	boolean
others	others	not correct	undetermined

où

- class1, class2 sont des classes dites "non-basic".
- class1 peut être :
    - la même que class2
    - une sous-classe de class2

4. L'opérateur SETEQ :

Syntaxe :

expr1 **SETEQ** *expr2*

Description :

arité : 2

position : infixe

associativité :

1. < prior(opérateur)
2. < prior(opérateur)

priorité : 3

arguments :

1. une expression PCL
2. une expression PCL

type : boolean

sémantique :

L'expression *expr1* **SETEQ** *expr2* prend la valeur :

- TRUE si l'ensemble de valeurs définis par l'expression *expr1* est le même ensemble de valeurs que celui défini par l'expression *expr2* .
- FALSE si l'ensemble de valeurs définis par l'expression *expr1* n'est pas le même ensemble de valeurs que celui défini par l'expression *expr2* .

Resctrictions :

Dans le cas où le deuxième argument est une variable, celle-ci doit déjà avoir été définie et désigner un ensemble de valeurs.

On notera également que si le premier argument est une variable non déjà apparue, celle-ci est alors définie et prend la même valeur que le deuxième argument.

Concordance de type :

expr1	expr2	comp	result type
set(tuple)	set(tuple)	correct	boolean
set(class1)	set(class2)	correct	boolean
set(integer)	set(integer)	correct	boolean
set(real)	set(real)	correct	boolean
set(string)	set(string)	correct	boolean
others	others	not correct	undetermined

où

- class1, class2 sont des classes dites "non-basic".
- class1 peut être :
    - la même que class2
    - une sous-classe de class2

4.2.8. Opérateurs definissant des fonctions agrégées

1. L'opérateur AVG :

Syntaxe :

AVG *expr1*

Description :

arité : 1

position : préfixe

associativité :

1. <= prior(opérateur)

priorité : 5

argument :

1. une expression PCL

type : real

sémantique :

La valeur de l'expression **AVG** *expr1* est la moyenne des valeurs de l'ensemble défini par l'expression *expr1* .

Resctrictions :

Dans le cas où l'argument est une variable, celle-ci doit déjà avoir été définie et désigner un ensemble de valeurs.

Concordance de type :

expr1	comp	result type
set(integer) set(real) others	correct correct not correct	real real undetermined

2. L'opérateur MIN :

Syntaxe :

MIN *expr1*

Description :

arité : 1

position : préfixe

associativité :

1. <= prior(opérateur)

priorité : 5

argument :

1. une expression PCL

type : integer, real, ou string

sémantique :

La valeur de l'expression **MIN** *expr1* est le minimum des valeurs de l'ensemble défini par l'expression *expr1* .

Resctrictions :

Dans le cas où l'argument est une variable, celle-ci doit déjà avoir été définie et désigner un ensemble de valeurs.

Concordance de type :

expr1	comp	result type
set(integer) set(real) set(string) others	correct correct correct not correct	integer real string undetermined

3. L'opérateur MAX :

Syntaxe :

MAX *expr1*

Description :

arité : 1

position : préfixe

associativité :

1. <= prior(opérateur)

priorité : 5

argument :

1. une expression PCL

type : integer, real, ou string

sémantique :

La valeur de l'expression **MAX** *expr1* est le maximum des valeurs de l'ensemble défini par l'expression *expr1* .

Resctrictions :

Dans le cas où l'argument est une variable, celle-ci doit déjà avoir été définie et désigner un ensemble de valeurs.

Concordance de type :

expr1	comp	result type
set(integer) set(real) set(string) others	correct correct correct not correct	integer real string undetermined

4. L'opérateur SUM :

Syntaxe :

SUM *expr1*

Description :

arité : 1

position : préfixe

associativité :

1. <= prior(opérateur)

priorité : 5

argument :

1. une expression PCL

type : integer ou real

sémantique :

La valeur de l'expression **SUM** *expr1* est la somme des valeurs de l'ensemble défini par l'expression *expr1* .

Resctrictions :

Dans le cas où l'argument est une variable, celle-ci doit déjà avoir été définie et désigner un ensemble de valeurs.

Concordance de type :

expr1	comp	result type
set(integer) set(real) others	correct correct not correct	integer real undetermined

5. L'opérateur COUNT :

Syntaxe :

COUNT *expr1*

Description :

arité : 1

position : préfixe

associativité :

1. <= prior(opérateur)

priorité : 5

argument :

1. une expression PCL

type : integer

sémantique :

La valeur de l'expression **COUNT** *expr1* est le cardinal de l'ensemble défini par l'expression *expr1* .

Resctrictions :

Dans le cas où l'argument est une variable, celle-ci doit déjà avoir été définie et désigner un ensemble de valeurs.

Concordance de type :

expr1	comp	result type
set(tuple)	correct	integer
set(class1)	correct	integer
set(integer)	correct	integer
set(real)	correct	integer
set(string)	correct	integer
others	not correct	undetermined

où

- class1 est une classe dite "non-basic".

4.2.9. Opérateurs arithmétiques

1. L'opérateur PLUS :

Syntaxe :

expr1 **PLUS** *expr2*

Description :

arité : 2

position : infixe

associativité :

1. <= prior(opérateur)
2. < prior(opérateur)

priorité : 7

arguments :

1. une expression PCL
2. une expression PCL

type : integer ou real

sémantique :

La valeur de l'expression *expr1* **PLUS** *expr2* est le résultat de la somme de l'expression *expr1* et de l'expression *expr2*.

Resctrictions :

Dans le cas où l'un des arguments est une variable, celle-ci doit déjà avoir été définie et désigner une valeur simple.

Concordance de type :

expr1	expr2	comp	result type
integer	integer	correct	integer
integer	real	correct	real
real	integer	correct	real
real	real	correct	real
others	others	not correct	undetermined

2. L'opérateur MINUS :

Syntaxe :

expr1 **MINUS** *expr2*

Description :

arité : 2

position : infixe

associativité :

1. <= prior(opérateur)
2. < prior(opérateur)

priorité : 7

arguments :

1. une expression PCL
2. une expression PCL

type : integer ou real

La valeur de l'expression *expr1* **MINUS** *expr2* est le résultat de la différence de l'expression *expr1* et de l'expression *expr2*.

Resctrictions :

Dans le cas où l'un des arguments est une variable, celle-ci doit déjà avoir été définie et désigner une valeur simple.

Concordance de type :

expr1	expr2	comp	result type
integer	integer	correct	integer
integer	real	correct	real
real	integer	correct	real
real	real	correct	real
others	others	not correct	undetermined

3. L'opérateur TIMES :

Syntaxe :

expr1 **TIMES** *expr2*

Description :

arité : 2

position : infixe

associativité :

1. <= prior(opérateur)
2. < prior(opérateur)

priorité : 6

arguments :

1. une expression PCL
2. une expression PCL

type : integer ou real

La valeur de l'expression *expr1* **TIMES** *expr2* est le résultat de la multiplication de l'expression *expr1* par l'expression *expr2* .

Resctrictions :

Dans le cas où l'un des arguments est une variable, celle-ci doit déjà avoir été définie et désigner une valeur simple.

Concordance de type :

expr1	expr2	comp	result type
integer	integer	correct	integer
integer	real	correct	real
real	integer	correct	real
real	real	correct	real
others	others	not correct	undetermined

4. L'opérateur DIV :

Syntaxe :

expr1 **DIV** *expr2*

Description :

arité : 2

position : infixe

associativité :

1. <= prior(opérateur)
2. < prior(opérateur)

priorité : 6

arguments :

1. une expression PCL
2. une expression PCL

type : real

La valeur de l'expression *expr1* **DIV** *expr2* est le résultat de la division de l'expression *expr1* par l'expression *expr2*.

Resctrictions :

Dans le cas où l'un des arguments est une variable, celle-ci doit déjà avoir été définie et désigner une valeur simple.

Concordance de type :

expr1	expr2	comp	result type
integer	integer	correct	real
integer	real	correct	real
real	integer	correct	real
real	real	correct	real
others	others	not correct	undetermined

4.3. Mot défini

Le mot-clé **THIS** est un mot défini pour le langage PCL. Il permet de faire référence à la première classe prise en considération pour l'expression. Il peut donc être employé partout où un nom de classe est autorisé.

Ce mot-clé est utilisé notamment dans la définition d'un slot, en permettant ainsi de désigner la classe pour laquelle le slot est défini.

Restrictions et extensions de la syntaxe abstraite

Annexe 9

1. Introduction

La syntaxe abstraite et la syntaxe concrète du langage ayant été intentionnellement décrites indépendamment l'une de l'autre, il est intéressant de présenter le parallélisme existant entre les deux types de syntaxes.

Rappelons que la syntaxe concrète n'est qu'une des représentations possibles pour la syntaxe abstraite.

Une première partie présente le parallélisme entre les expressions concrètes et les expressions abstraites.

Le point suivant reprend les restrictions de la syntaxe abstraite qui ont dû être appliquées pour implémenter la syntaxe concrète.

Un dernier point décrit les extensions "possibles" (de la syntaxe) du langage.

2. Parallélisme syntaxe abstraite / concrète

Cette notion de parallélisme est assez évidente, et sans intérêt pour les expressions primitives, ainsi que pour les expressions plus complexes telles les ensembles explicites de valeurs et les tuples.

Il est plus intéressant de comparer les opérateurs abstraits et concrets correspondants.

2.1. Parallélisme pour les expressions de base

Les expressions primitives de la syntaxe concrète, à l'exception de la variable, sont une simple représentation des expressions abstraites moyennant des conventions d'écriture.

En effet, ces conventions d'écriture, présentées dans l'annexe 8, sont les suivantes :

- Les constantes entières et réelles sont représentées telles quelles.
- Les constantes représentant des chaînes de caractères sont entourées de guillemets.
- Tout nom de classe commence par une lettre majuscule.
- Tout nom de slot commence par une lettre minuscule.
- Les expressions composées, telles les ensembles explicites de valeurs et les tuples, sont représentées entre les signes [et] .

2.2. Parallélisme pour les opérateurs

Pour chaque opérateur de la syntaxe concrète, on donnera l'opérateur abstrait correspondant, ainsi que la liste éventuelle des restrictions de la syntaxe abstraite introduites dans la syntaxe concrète pour cet opérateur.

1. opérateur # :

- ISofClass
- Aucune restriction

2. opérateur ? :

- aucun opérateur abstrait correspondant, une variable abstraite se distingue en tant que variable par la valeur de l'attribut object(variable) = var.
- Aucune restriction n'a été posée par rapport à la syntaxe abstraite pour la formulation d'une variable.

3. opérateur SETOF :

- SET
- Aucune restriction

4. opérateur WHERE :

- COND
- Aucune restriction

5. opérateur NOT, AND, OR :

- respectivement NOT, AND, OR
- Aucune restriction

6. opérateur EXIST ... MEMBER ... WITH :

- EXIST-MEMBER

- Aucune restriction n'a été posée par rapport à la syntaxe abstraite.

  Cependant, on remarquera que l'opérateur abstrait est d'arité 3. L'opérateur concret correpondant est toujours un opérateur d'arité 3, comportant trois mots-clé entourant les opérandes.

  Un tel opérateur ne peut être implémenté comme tel en PROLOG. Il faut utiliser des définitions intermédiaires d'opérateurs PROLOG.

  L'implémentation de cet opérateur est réalisée de la manière suivante :

  - opérateur (PROLOG) **EXIST** d'arité 1 :

    EXIST *expr1*

    où *expr1* est une expression de la forme

    expr2 **WITH** *expr3*

  - opérateur (PROLOG) **WITH** d'arité 2 :

    expr2 **WITH** *expr3*

    où *expr2* est une expression de la forme

    expr4 **MEMBER** *expr5*,

  - opérateur (PROLOG) **MEMBER** d'arité 2.

Notons :

Les opérateurs :

EXIST ... INCLUDED ... WITH,

FORALL ... MEMBER ... WITH,

FORALL ... INCLUDED ... WITH,

sont implémentés de la même manière.

7. opérateurS EQ, NE, GT, GE, ST, SE :

- respectivement EQ, NE, GT, GE, ST, GE
- Aucune restriction

8. opérateur MEMBER, INCLUDED :

- respectivement Member, Included

- Aucune restriction

9. opérateur ISIN :

- ISIN
- Aucune restriction

10. opérateur SETEQ :

- EqSet
- Aucune restriction

11. opérateurs AVG, MIN, MAX, SUM, COUNT :

- respectivement AVG, MIN, MAX, SUM, COUNT
- Aucune restriction

12. opérateur COUNT :

- COUNT
- Aucune restriction n'a été posée par rapport à la syntaxe abstraite.

  Cependant, on remarquera qu'il n'est pas possible d'obtenir le nombre d'instances d'une classe au moyen de l'opérateur COUNT comme on aurait pu l'espérer, ce dernier n'acceptant comme argument qu'une expression de type set(...), alors qu'un nom de classe n'est pas de type set(...) .

13. opérateurs PLUS, MINUS, TIMES, DIV :

- respectivement PLUS, MINUS, TIMES, DIV
- Aucune restriction

3. Restriction de la syntaxe abstraite

Une remarque préliminaire concerne la définition des variables. En effet, une variable de type integer, real, string, tuple, set(X) ne peut être définie que par l'intermédiaire d'un opérateur EQ ou SETEQ (assignation d'une valeur)

Une variable référençant une classe peut, quant à elle peut être définie au moyen de l'opérateur ISIN (définition de type) et peut ainsi être évaluée à l'une ou l'autre des instances de la classe, ou par l'intermédiaire de l'opérateur EQ (assignation de valeur).

Un premier type de restrictions porte sur l'utilisation des variables dans les expressions PCL.

Ces restrictions concernent les variables référençant des objets de type integer, real, string, tuple, set(....).

Une variable d'un tel type ne peut être utilisée dans une expression que si elle a été préalablement définie (dans la partie gauche de l'expression globale) au moyen d'une assignation de valeur par l'un des opérateurs EQ (pour les valeurs simples) ou SETEQ (pour les valeurs de type set(...)).

Cette restriction ne porte pas sur les variables référençant une classe, celles-ci pouvant être également définies au moyen de l'opérateur ISIN.

Une autre restriction de la syntaxe abstraite concerne les deux opérateurs abstraits suivants :

- EXIST-ISIN
- FORALL-ISIN

Ces opérateurs abstraits n'ont en effet pas de correspondant dans la syntaxe concrète. Ils n'ont pu être implémentés, suite aux restrictions portant sur l'usage des variables.

4. Extensions possibles

Une extension de la syntaxe abstraite est l'apparition dans la syntaxe concrète d'un opérateur permettant de désigner une variable. La description de cet opérateur est présentée dans l'annexe 8 concernant la syntaxe concrète.

Cependant, ce type d'extension n'accroît pas les possibilités du langage PCL.

Les caractéristiques suivantes, quant à elles, permettent d'étendre les possibilités du langage PCL.

Mais ces extensions peuvent également entraîner des modifications de la syntaxe abstraite du langage.

Notamment, l'ajout d'attribut pour la grammaire attribuée vérifiant la corection syntaxique des expressions, la modification de certaines règles syntaxiques, ... peuvent devoir être envisagés.

Une première extension possible serait de considérer qu'un ensemble explicite de valeurs ne soit plus restreint à un ensemble de constantes, mais puisse également désigner un ensemble d'identifiants d'instances de classes dites non-basic.

La définition abstraite d'un tel ensemble serait :

$[c_1, \ldots, c_n]$

où

- $n >= 0$
- $(\forall\ i : 1<=i<=n)$: soit
  - c_i est une constante,
  - c_i est un identifiant d'instance.
- $\forall\ i, j$: c_i et c_j sont de même type

Cette extension de la définition des ensembles explicites de valeurs pose le problème de la distinction d'un tel ensemble d'identifiants d'instances et de l'occurrence d'un tuple.

Il faudrait donc modifier la grammaire attribuée pour la correction syntaxique des expressions, de manière à pouvoir tenir compte de cette distinction.

Une seconde extension des possibilités du langage serait de permettre au programmeur d'introduire dans une expression PCL un appel à une procédure PROLOG qu'il aurait écrite.

Cet appel de procédure PROLOG pourrait être de deux types :

- Un appel de procédure ayant pour effet d'attribuer une valeur à une variable,
- Un appel de procédure ayant pour effet de vérifier une contrainte, et de renvoyer ainsi une valeur booléenne.

Une telle extension introduit immédiatemment le problème de la correction syntaxique d'une expression contenant un tel appel.

En effet, dans le cas d'un appel de procédure ayant pour effet d'attribuer une valeur à une variable, il est impossible de déterminer le type de la variable avant l'évaluation proprement dite de l'expression (avant l'exécution de la procédure), car le langage PROLOG est non-typé.

Toute vérification syntaxique portant sur les types devient donc impossible dans cette optique.

Une solution est alors de ne vérifier syntaxiquement que les expressions pour lesquelles aucun problème n'apparaît. Dans les autres cas, aucune vérification syntaxique (autre que celles effectuées par le PROLOG reader) n'est effectuée avant l'évaluation.

Choix d'implémentation

Annexe 10

1. Introduction

Ce document donne un aperçu de la méthode suivie pour l'implémentation de la syntaxe concrète.

On trouvera également une liste des codes d'erreurs fournis par les procédures.

2. Définition "Prolog" des opérateurs

Partant de la description des operateurs de la syntaxe concrète dans l'annexe 8, il est relativement aisé d'en déduire une définition PROLOG pour chaque opérateur.

Rappelons que les principales caractéristiques intervenant dans la définition d'un opérateur sont les suivantes:

- nom de l'opérateur
- arité
- niveau de priorité: un entier compris entre 0 et 1200
- règles d'associativité: xfx, xfy, yfx, fy, fx, xf, yf

  Ces règles prennent en compte les caractéristiques de position et d'associativité de l'opérateur.

Pour chaque opérateur, on donnera son arité, son niveau de priorité, et les règles d'associativité sous la forme:

OP(*niv*, *ass*, *op* **)**

où *niv* est le niveau de priorité, *ass* les règles de priorité, et *op* le nom de l'opérateur.

```
OP(670, xfy, OR)
OP(660, xfy, AND)
OP(650, fy, NOT)
OP(640, yfx, PLUS)
OP(640, yfx, MINUS)
OP(630, yfx, TIMES)
OP(630, yfx, DIV)

OP(620, fy, SETOF)
OP(620, fy, FORALL)
OP(620, fy, EXIST)
OP(620, fy, AVG)
OP(620, fy, MIN)
OP(620, fy, MAX)
OP(620, fy, SUM)
OP(620, fy, COUNT)

OP(610, xfy, WHERE)
OP(610, xfy, WITH)

OP(600, xfx, EQ)
OP(600, xfx, NE)
OP(600, xfx, GT)
OP(600, xfx, GE)
OP(600, xfx, ST)
OP(600, xfx, SE)
OP(600, xfx, MEMBER)
OP(600, xfx, INCLUDED)
OP(600, xfx, ISIN)
OP(600, xfx, SETEQ)

OP(90, xfy, # )
OP(80, fx, ? )
```

3. Implémentation de la syntaxe concrète

L'implémentation du langage PCL, comme elle a été présentée au chapitre 3 de la quatrième partie, consiste en l'implémentation de deux composants réalisant ainsi l'interface avec le langage PROBE.

Pour rappel, on distinguera les composants suivants:

- Le composant d'analyse syntaxique
- Le composant d'évaluation

Les points suivants présenteront:

- Une classification de l'ensemble des procédures PROLOG qui sont nécessaires à la réalisation des différents interfaces.
- Le schéma de spécification d'une procédure PROLOG qui a été utilisé (sur base de [DEV]).
- Une série de remarques générales valables pour les différentes spécifications.
- Une présentation des différentes procédures, suivant la classification proposée.

  Cette présentation, selon le degré de complexité de la procédure, se limite à sa spécification, ou expose un certain nombre de précisions concernant son implémentation.
- Une présentation des codes d'erreur renvoyés par les procédures.

3.1. Classification des procédures

Les procédures peuvent être regroupées en trois grands types:

- procédures d'interface:

  reprenant les procédures réalisant l'interface avec le langage PROBE (composant d'analyse syntaxique et composant d'évaluation).
- procédures primitives:

  reprenant les procédures déduites des grammaires attribuées, et nécessaires à l'implémentation des procédures réalisant l'interface.
- procédures de manipulation:

reprenant des procédures à usage général, principalement des procédures de manipulation de listes.

3.2. Schéma de spécification d'une procédure

Le schéma de spécification d'une procédure PROLOG est décrit par le plan suivant, inspiré de [DEV] et complété par des informations utiles dans notre cas:

Procédure: nom_de_procédure_p(p_1, ... ,p_n)

Type:

Pour chaque argument p_i: une description aussi précise que possible de son type.

Relation:

Une description précise de la relation liant les arguments de la procédure

Directionalité:

Selon le schéma:

in(dp_1, ... , dp_n): out(dp_1, ... , dp_n),

où dp_i indique la directionalité de l'argument p_i,

à savoir:

ground ou totalement instancié (g),
variable (v),
partiellement instancié (ngv),
ground ou variable (gv),
ground ou partiellement instantié (novar),
sans restriction (any).

Multiplicité:

Elle indique les nombres minimum et maximum de substitutions résultats que la procédure peut fournir, sous la forme ‹min, max› .

Nature:

Elle permet de situer la procédure sur base de la classification présentée ci-dessus.

On distinguera donc la nature par l'une des valeurs suivantes:

- interface
- primitive

- manipulation

Préconditions sur l'environnement:

Dans le cas où certaines préconditions (toute condition autre que celles portant sur les arguments) sur l'environnement sont nécessaires, celles-ci doivent être énoncées.

Effets de bord:

Toute modification de l'environnement est précisé.

Codes d'erreurs:

Dans le cas où un code d'erreur est retourné par la procédure, on précisera la nature et le positionnement de ce code.

3.3. Remarques générales

- Toute procédure de nature "interface" doit satisfaire la précondition suivante:

  "Un fichier, contenant la Base de Connaissances particulière pour laquelle une expression va être analysée ou évaluée, doit être ouvert".

  Cette précondition ne sera pas reprise pour chaque procédure de nature "interface".

- La directionalité (et la multiplicité) présentées dans la spécification d'une procédure correspondent à celles qui étaient nécessaires à l'usage attendu de cette procédure.

  Ceci ne signifie pas nécessairement que cette procédure ne peut fonctionner avec d'autres directionalités. Simplement, ces autres directionalités n'ont pas été prévues et ne sont donc pas garanties.

3.4. Présentation des procédures

Ce point présente les différentes procédures, suivant la classification proposée au point 3.1.

3.4.1. Procédures de manipulation

Procédure: appendlist(_list1, _list2, _lresult)

Type:

_list1, _list2, _lresult: liste

Relation:

la liste _lresult est la concaténation des deux listes _list1 et _list2.

Directionalité: in(g,g,gv): out(g,g,g)

Multiplicité: <0, 1>

Nature: manipulation

Procédure: is_alist(_term)

Type:

_term: terme

Relation:

Le terme _term est une liste.

Directionalité: in(g): out(g)

Multiplicité: <0, 1>

Nature: manipulation

Procédure: is_first(_el, _list)

Type:

_el: terme

_list: liste de termes

Relation:

la liste _list est non vide, et _el est son dernier élément.

Directionalité: in(gv,g): out(g,g)

Multiplicité: <0, 1>

Nature: manipulation

Procédure: is_inlist(_el, _list)

Type:

_el: terme

_list: liste de termes

Relation:

la liste _list contient l'element _el.

Directionalité: in(g,g): out(g,g)

Multiplicité: <0, 1>

Nature: manipulation

Procédure: is_partoflist(_sublist, _list)

Type:

_sublist, _list: liste de termes

Relation:

Tout élément de la liste _sublist est élément de la liste _list.

Directionalité: in(g,g): out(g,g)

Multiplicité: <0, 1>

Nature: manipulation

Procédure: equal_list(_list1, _list2)

Type:

_list1, _list2: liste de termes

Relation:

les listes _list1 et _list2 ont même longueur, et contiennent les mêmes éléments (pas nécessairement dans le même ordre).

Directionalité: in(g,g): out(g,g)

Multiplicité: ‹0, 1›

Nature: manipulation

Procédure: suppress(_el,_list, _listres)

Type:

_el: terme

_list, _listres: liste de termes

Relation:

la liste _listres est la liste _list, sans la première occurrence de l'élément _el.

Directionalité: in(g,g,gv): out(g,g,g,)

Multiplicité: ‹0, 1›

Nature: manipulation

3.4.2. Procédures primitives

Procédure: give_class(_id, _cl)

Type:

_id: de la forme "atome PROLOG / integer"

_cl: atome PROLOG

Relation:

_id est un identifiant d'instance correct, et _cl est le nom de classe qu'il contient.

Directionalité: in(any, gv): out(g,g)

Multiplicité: ‹0, *›

Nature: primitive

Préconditions sur l'environnement:

"Un fichier, contenant la Base de Connaissances particulière pour laquelle une expression va être analysée ou évaluée, doit être ouvert".

Effets de bord:

Consultation du fichier contenant la Base de Connaissances, et mise-à-jour du code d'erreur lié aux prédicats PROBE.

Codes d'erreurs:

Dans le cas où la première partie (atome) de l'identifiant n'est pas un nom de classe du modèle conceptuel, un code d'erreur est mis à jour par le prédicat PROBE chargé de consulter la Base de Connaissances.

Sinon, ce code d'erreur retourne la valeur indiquant la réussite du prédicat.

Procédure: classtype(_classname, _type)

Type:

_classname: atome

_type: atome

Relation:

_classname est le nom d'une classe du modèle conceptuel, de type PCL _type.

Directionalité: in(gv, gv): out(g, g)

Multiplicité: ‹0, 1›

Nature: primitive

Préconditions sur l'environnement:

"Un fichier, contenant la Base de Connaissances particulière pour laquelle une expression va être analysée ou évaluée, doit être ouvert".

Effets de bord:

Consultation du fichier contenant la Base de Connaissances, et mise-à-jour du code d'erreur lié aux prédicats PROBE.

Codes d'erreurs:

Dans le cas où _classname n'est pas un nom de classe du modèle conceptuel, un code d'erreur est mis à jour par le prédicat PROBE chargé de consulter la Base de Connaissances.

Sinon, ce code d'erreur retourne la valeur indiquant la réussite du prédicat.

Procédure: is_aclass(_cl)

Type:

_cl: atome

Relation:

_cl est le nom d'une classe de la Base de Connaissances.

Directionalité: in(g): out(g)

Multiplicité: <0, 1>

Nature: primitive

Préconditions sur l'environnement:

"Un fichier, contenant la Base de Connaissances particulière pour laquelle une expression va être analysée ou évaluée, doit être ouvert".

Effets de bord:

Consultation du fichier contenant la Base de Connaissances, et mise-à-jour du code d'erreur lié aux prédicats PROBE.

Codes d'erreurs:

Dans le cas où _cl n'est pas un nom de classe du modèle conceptuel, un code d'erreur est mis à jour par le prédicat PROBE chargé de consulter la Base de Connaissances.

Sinon, ce code d'erreur retourne la valeur indiquant la réussite du prédicat.

Procédure: is_aslot(_slot)

Type:

_slot: atome

Relation:

_slot est le nom d'un slot défini pour une classe de la Base de Connaissances.

Directionalité: in(g): out(g)

Multiplicité: <0, 1>

Nature: primitive

Préconditions sur l'environnement:

"Un fichier, contenant la Base de Connaissances particulière pour laquelle une expression va être analysée ou évaluée, doit être ouvert".

Effets de bord:

Consultation du fichier contenant la Base de Connaissances, et mise-à-jour du code d'erreur lié aux prédicats PROBE.

Codes d'erreurs:

Dans le cas où _slot n'est pas un nom de slot du modèle conceptuel, un code d'erreur est mis à jour par le prédicat PROBE chargé de consulter la Base de Connaissances.

Sinon, ce code d'erreur retourne la valeur indiquant la réussite du prédicat.

Procédure: is_aconstant(_cst)

Type:

_cst: atome

Relation:

la valeur _cst définit bien une constante PCL.

Directionalité: in(g): out(g)

Multiplicité: ‹0, 1›

Nature: primitive

Procédure: is_atuple(_term)

Type:

_term: liste

Relation:

le terme _term définit bien un tuple au sens PCL.

Directionalité: in(g): out(g)

Multiplicité: ‹0, 1›

Nature: primitive

Procédure: is_asetofvalues(_term)

Type:

_term: liste

Relation:

le terme _term définit bien un ensemble explicite de valeurs.

Directionalité: in(g): out(g)

Multiplicité: ‹0, 1›

Nature: primitive

Procédure: class_refer(_term, _cl)

Type:

_term: terme

_cl: atome

Relation:

la classe _cl est la classe référencée par l'expression PCL _term

Directionalité: in(g, gv): out(g, g)

Multiplicité: ‹0, 1›

Nature: primitive

Préconditions sur l'environnement:

"Un fichier, contenant la Base de Connaissances particulière pour laquelle une expression va être analysée ou évaluée, doit être ouvert".

Effets de bord:

Consultation du fichier contenant la Base de Connaissances, et mise-à-jour du code d'erreur lié aux prédicats PROBE.

Codes d'erreurs:

Dans le cas où _term contient soit des noms de classes, soit des noms de slots qui ne sont pas décrits dans le modèle conceptuel, un code d'erreur est mis à jour par le prédicat PROBE chargé de consulter la Base de Connaissances.

Sinon, ce code d'erreur retourne la valeur indiquant la réussite du prédicat.

Procédure: result_type(_term, $_type_1$, ..., $_type_n$, _resulttype)

Type:

_term: atome

$_type_1$, ..., $_type_n$, _resulttype: atome

Relation:

Les différentes procédures correspondant à l'en-tête sont l'implémentation des tables définissant les règles de concordances de type pour chaque opérateur. Ces tables sont définies pour la grammaire attribuée définissant la syntaxe abstraite.

Directionalité: in(g, g, ..., g, gv): out(g, g, ..., g, g)

Multiplicité: ‹0, 1›

Nature: primitive

Préconditions sur l'environnement:

"Un fichier, contenant la Base de Connaissances particulière pour laquelle une expression va être analysée ou évaluée, doit être ouvert".

Effets de bord:

Consultation du fichier contenant la Base de Connaissances, et mise-à-jour du code d'erreur lié aux prédicats PROBE.

Codes d'erreurs:

Dans le cas où l'un des atomes $_type_i$ désigne un type PCL référençant un nom de classe qui n'est pas décrit dans modèle conceptuel, un code d'erreur est mis à jour par le prédicat PROBE chargé de consulter la Base de Connaissances.

Sinon, ce code d'erreur retourne la valeur indiquant la réussite du prédicat.

3.4.3. Procédures d'interface

3.4.3.1. Composant d'analyse syntaxique

La procédure réalisant cet interface vérifie la correction syntaxique d'une expression et fournit la liste des relations de dépendances pour l'expression.

A. Spécification de la procédure principale

Procédure: evaldepend(_term, _cl, _list)

Type:

_term: terme

_cl: atome

_list: liste de termes

Relation:

La liste _list est la liste des relations de dépendances de l'expression PCL _term, et _cl est un nom de classe pouvant être référencé par des noms de slots, de variables apparaissant dans l'expression _term.

Directionalité: in(g, gv, var): out(g, gv, g)

Multiplicité: <0, 1>

Nature: interface

Codes d'erreurs:

La variable PCLrc est mise à jour.

Description intuitive des arguments:

1. _term:

Le paramètre _term est l'expression à analyser (vérifier que l'expression PCL est correcte syntaxiquement), et pour laquelle il faut fournir les relations de dépendances.

(Paramètre devant être instancié au moment de l'appel.)

2. _cl:

Le paramètre _cl est un nom de classe pouvant apparaître dans l'expression. Le mot-clé THIS apparaissant dans l'expression désigne cette classe.

(Paramètre pouvant être instancié au moment de l'appel.)

3. _list:

Le paramètre _list est la liste des relations de dépendances déduites de l'expression.

(Paramètre ne pouvant pas être instancié au moment de l'appel.)

B. Implémentation

L'implémentation de la procédure principale evaldepend(_term, _cl, _list) consiste en l'implémentation d'une procédure de nom evalD, cette dernière ayant un plus grand nombre d'arguments non significatifs pour le monde extérieur, mais utiles du point de vue de l'implémentation.

1. Sous-procédure evalD

a. Spécification

Procédure: evalD(_term, _inhref, _nature, _state)

Type:

_term: terme

_inhref: liste d'éléments de forme "class(at_1)" ou "variable(at2, at3, at4)", où :

at1 est un atome désignant un nom de classe,
at2 est un atome désignant une variable PCL,
at3 est un type au sens PCL,
at4 est un atome désignant un nom de classe.

_nature: l'une des valeurs exp, cond, class, var, tuple

_state: terme PROLOG de la forme "state(_type,_ref,_list)", où

_typ est un atome désignant un nom de classe,
_ref est une liste de même type que _inhref,
_list est une liste.

Relation:

L'expression PCL _term hérite d'une liste _inhref de classes et variables pouvant apparaître dans l'expression, est de nature _nature et de type _type, contient les noms de classes et variables de la liste _ref, et est caractérisée par les relations de dépendances de la liste _list.

Directionalité: in(g,g,gv,ngv): out(g,g,g,g)

Multiplicité: ‹0, 1›

Nature: interface

Codes d'erreurs:

La variable PCLrc est mise à jour.

Description intuitive des arguments:

1. _term:

   Le paramètre _term est l'expression à analyser (vérifier que l'expression PCL est correcte syntaxiquement), et pour laquelle il faut fournir les relations de dépendances.

   (Paramètre devant être instancié au moment de l'appel.)

2. _inhref: (pour "inherited referent")

   Le paramètre _inhref est une liste de termes de la forme "class(classname)" ou "var(_var, _type, _clref)", reprenant les noms de classes ou de variables pouvant apparaître dans l'expression.

   On remarquera qu'à une variable doivent également être associés un type et la classe qu'elle référence.

   (Paramètre devant être instancié au moment de l'appel.)

   Ce paramètre implémente en une seule liste les attributs hérités "inh-ref" et "inh-tabVar" de la grammaire attribuée pour la correction syntaxique des expressions.

3. _nature:

   Le paramètre _nature permet de caractériser l'expression. Il est l'un des mots suivants:

   - exp (pour expression).
   - cond (pour condition booléenne).
   - class (pour classe).

- var (pour variable).
- tuple (pour tuple).

(Paramètre pouvant être instancié au moment de l'appel.)

Ce paramètre implémente l'attribut "nature" de la grammaire attribuée pour la correction syntaxique des expressions.

4. _state:

Le paramètre _state est un terme prolog (abstraction de données) de la forme "state(_type, _ref, _list)" où :

- _type est le type résultant de l'expression.

  Ce paramètre implémente l'attribut "type" de la grammaire attribuée pour la correction syntaxique des expressions.

- _ref est la liste des classes et variables apparues dans l'expression (de même forme que _inhref).

  Ce paramètre implémente en une seule liste les attributs synthétisés "ref" et "tabVar" de la grammaire attribuée pour la correction syntaxique des expressions.

- _list est la liste des relations de dépendances pour l'expression.

  Ce paramètre implémente l'attribut défini dans l'annexe 6.

(Paramètre ne pouvant être complètement instancié au moment de l'appel: le type peut être instancié, contrairement aux deux autres paramètres.)

b. Description intuitive

L'implémentation de la procédure

evalD(_term,_inhref, _nature, _state)

est basée sur l'algorithme suivant:

- reconnaissance de l'opérateur principal composant l'expression _term:

  Cette étape permet de déterminer chacun des opérandes.

- exécution de la procécure correspondant à l'opérateur reconnu et analysant chacun des opérandes déterminés

Ce type d'implémentation permet de limiter l'analyse d'une expression à la reconnaissance de l'opérateur principal. Il induit également la construction d'une procédure par opérateur.

c. Algorithme

L'algorithme de la procédure evalD(_term, _inhref, _nature, _state) est le suivant:

Si _term = opérateur1(arg_1, ..., arg_n)

ALORS

analyserOpérateur1(arg_1, ..., arg_n,_inhref, _nature,_state)

SINON SI _term = opérateur2(arg_1, ..., arg_k)

ALORS

analyserOpérateur2(arg_1, ..., arg_k,_inhref, _nature,_state)

SINON SI ....

....

SINON SI _term = opérateurJ(arg_1, ..., arg_m)

ALORS

analyserOpérateurJ(arg_1,....., arg_i,_inhref, _nature,_state)

SINON signalerErreur.

2. Sous-procédure d'analyse d'un opérateur particulier

Une description générale d'une procédure analyserOpérateur peut être fournie:

Description:

procédure **analyserOpérateurJ**(arg_1,.....,arg_i,_inhref,_n, _s)

De manière générale, on peut dire que chaque procédure correspondant à un opérateur analyse chacun des opérandes déterminés, et vérifie la concordance de type.

La procédure de nom "analyserOpérateurJ" est implémentée de la manière suivante:

- au moment de l'appel, les (i + 1) premiers paramètres doivent être instanciés.
- analyse des (i) premiers paramètres en tant qu'expressions PCL (appel de la procédure de nom evalD sur des expressions plus petites).
- consultation des règles de concordances de types, sur base de la grammaire abstraite.
- construction du terme prolog state(_type, _ref, _list):
    - _type est le type de l'expression, obtenu après consultation des règles de concordances de types (au moyen de la procédure prolog "result_type").
    - _ref est la liste construite à partir du paramètre "_inhref" en respectant les règles de construction données par les attributs "inh-ref","ref","inh-tabVar" et "tabVar" de la grammaire attribuée pour la syntaxe abstraite.
    - _list est la liste des relations de dépendances.

3.4.3.2. Composant d'évaluation

Cette procédure évalue une expression syntaxiquement correcte et fournit la valeur de cette expression.

A. Spécification de la procédure principale

Procédure: evaluation(_term, _ins, _value)

Type:

_term: terme

_ins: de la forme "atome PROLOG / integer"

_value: élément ou liste d'éléments selon le cas

Relation:

La valeur _value est le résultat de l'évaluation de l'expression PCL _term syntaxiquement correcte, celle-ci étant évaluée pour l'instance désignée par _ins.

Directionalité: in(g, gv, var): out(g, gv, g)

Multiplicité: ‹0, *›

Nature: interface

Codes d'erreurs:

La variable PCLrc est mise à jour.

Description intuitive des arguments:

1. _term:

   Le paramètre _term est l'expression syntaxiquement correcte à évaluer, et pour laquelle il faut fournir la valeur.

   (Paramètre devant être instancié au moment de l'appel.)

2. _ins:

   Le paramètre _ins est un identifiant d'instance d'une classe, pour laquelle l'expression doit être évaluée.

   (Paramètre pouvant être instancié au moment de l'appel.)

3. _value:

   Le paramètre _value est la valeur de l'expression.

   (Paramètre ne pouvant pas être instancié au moment de l'appel.)

B. Implémentation

L'implémentation de la procédure evaluation(_term, _ins, _value) consiste en l'implémentation d'une procédure de nom evalV, cette dernière ayant un plus grand nombre d'arguments non significatifs pour le monde extérieur, mais utiles du point de vue de l'implémentation.

1. Sous-procédure evalV

a. Spécification

evalV(_term, _inhstate, _nature, _state, _value)

Description intuitive des arguments:

1. _term:

   Le paramètre _term est l'expression syntaxiquement correcte à évaluer, et pour laquelle il faut fournir la valeur.

   (Paramètre devant être instancié au moment de l'appel.)

2. _inhstate:

Le paramètre _inhstate est un terme prolog (abstraction de données) de la forme "inh(_inhr, _inht, _ins)" où:

- _inhr est une liste d'éléments de la forme "class(classname)" ou "var(_var, _type, _clref)", reprenant les noms de classes ou de variables pouvant apparaître dans l'expression.

  Ce paramètre implémente en une seule liste les attributs hérités "inh-ref" et "inh-tabVar" de la grammaire attribuée pour la correction syntaxique des expressions.

- _inht est une liste d'éléments de la forme "cl(classname, _val)" ou "var(_var, _type, _val)", reprenant les noms de classes ou de variables pouvant apparaître dans l'expression, ainsi que leur valeur.

  Ce paramètre implémente l'attribut hérité "inh-Table" de la grammaire attribuée pour la sémantique des expressions.

- _ins est un identifiant d'instance d'une classe, pour laquelle l'expression doit être évaluée.

  Ce paramètre implémente l'attribut "insForEval" de la grammaire attribuée pour la sémantique des expressions.

On remarquera qu'à une variable doit également être associé un type.

(Paramètre ne pouvant être complètement instancié au moment de l'appel: l'argument _ins peut ne pas être instancié, contrairement aux deux autres paramètres.)

3. _nature:

Le paramètre _nature permet de caractériser l'expression. Il est l'un des mots suivants:

- exp (pour expression).
- cond (pour condition booléenne).
- class (pour classe).
- var (pour variable).
- tuple (pour tuple).

(Paramètre pouvant être instancié au moment de l'appel.)

Ce paramètre implémente l'attribut "nature" de la grammaire attribuée pour la sémantique des expressions.

4. _state:

Le paramètre _state est un terme prolog de la forme "st(_type, _r, _t)" où :

- _type est le type résultant de l'expression,

  Ce paramètre implémente l'attribut "type" de la grammaire attribuée pour la correction syntaxique des expressions.

- _r est la liste des classes et variables apparues dans l'expression (de même forme que _inhr), et

  Ce paramètre implémente en une seule liste les attributs synthétisés "ref" et "tabVar" de la grammaire attribuée pour la correction syntaxique des expressions.

- _t est la liste des classes et variables apparues dans l'expression, ainsi que leur valeur (de même forme que _inht) .

  Ce paramètre implémente l'attribut synthétisé "Table" de la grammaire attribuée pour la sémantique des expressions.

(Paramètre ne pouvant être complètement instancié au moment de l'appel: le type peut être instancié, contrairement aux deux autres paramètres.)

5. _value:

Le paramètre _value est la valeur de l'expression PCL.

(Paramètre ne pouvant pas être instancié au moment de l'appel.)

Ce paramètre implémente l'attribut "eval" de la grammaire attribuée pour la sémantique des expressions.

b. Description intuitive

L'implémentation de la procédure

evalV(_term,_inhstate, _nature, _state, _value)

est basée sur l'algorithme suivant:

- reconnaissance de l'opérateur principal composant l'expression _term:

  Cette étape permet de déterminer chacun des opérandes.

- exécution de la procécure correspondant à l'opérateur reconnu et évaluant chacun des opérandes déterminés
- évaluation de l'expression à partir des valeurs de chacuns des opérandes

Ce type d'implémentation permet de limiter l'évaluation d'une expression à la reconnaissance de l'opérateur principal. Il induit également la construction d'une procédure par opérateur.

c. Algorithme

L'algorithme de la procédure evalV(_term,_inhstate, _nature, _state, _value) est le suivant:

Si _term = opérateur1(arg_1, ..., arg_n)

ALORS

evaluerOpérateur1(arg_1, ..., arg_n,_inh, _n,_st, _val)

SINON SI _term = opérateur2(arg_1, ..., arg_k)

ALORS

evaluerOpérateur2(arg_1, ..., arg_k,_inh, _n,_st, _val)

SINON SI ....

SINON SI _term = opérateurJ(arg_1, ..., arg_m)

ALORS

evaluerOpérateurJ(arg_1, ..., arg_i,_inh, _n,_st, _val)

SINON signalerErreur.

2. Sous-procédure d'analyse d'un opérateur particulier

Une description générale d'une procédure evaluerOpérateur peut être fournie:

Description:

procédure **evaluerOpérateurJ**(arg_1, ..., arg_i,_inh, _n,_st, _val)

De manière générale, on peut dire que chaque procédure correspondant à un opérateur évalue chacun des opérandes déterminés, vérifie la concordance de type, et fournit alors la valeur

globale de l'expression à partir des valeurs de chacuns des opérandes.

La procédure de nom "evaluerOpérateurJ" est implémentée de la manière suivante:

- au moment de l'appel, les (i + 1) premiers paramètres doivent être instanciés.
- évaluation des (i) premiers paramètres en tant qu'expressions PCL (appel de la procédure de nom evalV sur des expressions plus petites).
- consultation des règles de concordances de types, sur base de la grammaire abstraite.
- construction du terme Prolog st(_type, _r, _t):
    - _type est le type de l'expression, obtenu après consultation des règles de concordances de types (au moyen de la procédure prolog "result_type").
    - _r est la liste construite à partir du paramètre "_inhr" en respectant les règles de construction données par les attributs "inh-ref","ref","inh-tabVar" et "tabVar" de la grammaire attribuée pour la syntaxe abstraite.
    - _t est la liste construite à partir du paramètre "_inht" en respectant les règles de construction données par les attributs "inh-Table" et "Table" de la grammaire attribuée pour la sémantique.

3.5. Codes d'erreurs

Le contrôle d'erreur est géré de la manière suivante:

Toute erreur intervenant à l'une quelconque des phases d'analyse d'une expression PCL a pour effet la mise-à-jour d'une variable globale appelée PCLrc (pour "**PCL r**eturn **c**ode").

Cette variable globale **PCLrc** est mise à jour par l'une ou l'autre des procédures PROLOG d'entête :

- evaldepend(_term, _cl, _list).
- evaluation(_term, _ins, _value).

Toute modification de la variable globale PCLrc rend compte d'une erreur par rapport aux règles décrites dans les grammaires attribuées pour la correction syntaxique, et la sémantique d'une expression.

Cette variable est initialisée à chaque analyse d'une expression.

Si l'analyse de l'expression s'est terminée correctement, la variable globale PCLrc conserve sa valeur d'initialisation, à savoir PCL0. Dans le cas contraire, dès qu'une erreur est rencontrée, cette variable globale se voit assigner un code d'erreur.

Ce code peut être un code signalant une erreur survenue lors d'un accès à la Base de Connaissances au moyen d'un prédicat PROBE. Dans ce cas, le code d'erreur retourné est un entier. Dans le cas où l'erreur survenue est détectée par un composant implémentant le langage PCL,le code d'erreur est de la forme PCLi, où i est un entier.

La forme du code retourné dans ce cas permet de distinguer les erreurs détectées au niveau de PCL de celles détectées au niveau de PROBE.

D'autre part, le choix de numéroter les erreurs plutôt que de renvoyer un message plus significatif a été effectué dans un souci de garder une similitude avec les codes d'erreurs de PROBE.

Les différents codes d'erreurs propres à PCL sont:

- **PCLrc = PCL0**

  Aucune erreur n'a été rencontrée et la procédure s'est terminée correctement.

- **PCLrc = PCL2**

  Pas de classe référencée pour la variable, car variable non définie

- **PCLrc = PCL5**

  Expression non correct contenant l'opérateur #.

- **PCLrc = PCL6**

  Expression non correcte contenant l'opérateur # et pour laquelle le premier opérande est une variable.

- **PCLrc = PCL7**

  Expression non correcte contenant l'opérateur WHERE.

- **PCLrc = PCL8**

  Utilisation d'une variable non définie.

- **PCLrc = PCL9**

  Utilisation d'une classe non définie.

- **PCLrc = PCL10**

  Utilisation d'une expression qui n'est pas une constante.

- **PCLrc = PCL11**

  Type "undetermined" pour l'opérateur EQ.

- **PCLrc = PCL12**

  Expression non correcte contenant l'opérateur EQ.

- **PCLrc = PCL14**

  Type "undetermined" pour l'opérateur NE.

- **PCLrc = PCL13**

  Expression non correcte contenant l'opérateur NE.

- **PCLrc = PCL15**

  Type "undetermined" pour l'opérateur GT

- **PCLrc = PCL16**

  Expression non correcte contenant l'opérateur GT.

- **PCLrc = PCL17**

  Type "undetermined" pour l'opérateur GE.

- **PCLrc = PCL18**

  Expression non correcte contenant l'opérateur GE.

- **PCLrc = PCL19**

  Type "undetermined" pour l'opérateur ST.

- **PCLrc = PCL20**

  Expression non correcte contenant l'opérateur ST.

- **PCLrc = PCL21**

  Type "undetermined" pour l'opérateur SE.

- **PCLrc = PCL22**

  Expression non correcte contenant l'opérateur SE.

- **PCLrc = PCL23**

  Expression non correcte contenant l'opérateur NOT.

- **PCLrc = PCL24**

  Expression non correcte contenant l'opérateur AND.

- **PCLrc = PCL25**

  Expression non correcte contenant l'opérateur OR.

- **PCLrc = PCL27**

  Expression non correcte contenant le mot défini THIS, pas de classe pouvant être référencée par ce mot THIS.

- **PCLrc = PCL28**

  Expression non correcte contenant l'opérateur # et pour laquelle le premier opérande est une classe dite basic.

- **PCLrc = PCL29**

  Utilisation d'un slot non défini.

- **PCLrc = PCL30**

  Essai d'évaluation d'une variable pour une instance ne pouvant être une valeur pour cette variable.

- **PCLrc = PCL31**

  Essai d'évaluation d'une variable référençant une classe pour laquelle aucne instance n'existe.

- **PCLrc = PCL32**

  Essai d'évaluation d'une variable définie pour laquelle aucune valeur n'est fixée et qui aurait dû posséder une valeur.

- **PCLrc = PCL33**

  Expression non correcte contenant l'opérateur PLUS.

- **PCLrc = PCL34**

  Expression non correcte contenant l'opérateur MINUS.

- **PCLrc = PCL35**

  Expression non correcte contenant l'opérateur TIMES.

- **PCLrc = PCL36**

  Expression non correcte contenant l'opérateur DIV.

- **PCLrc = PCL37**

  Expression non correcte contenant l'opérateur SETOF.

- **PCLrc = PCL38**

  Incompatibilité de type pour l'opérateur SETOF.

- **PCLrc = PCL39**

  Incompatibilité de type pour l'opérateur AVG.

- **PCLrc = PCL40**

Incompatibilité de type pour l'opérateur MIN.

- **PCLrc = PCL41**

  Incompatibilité de type pour l'opérateur MAX.

- **PCLrc = PCL42**

  Incompatibilité de type pour l'opérateur SUM.

- **PCLrc = PCL43**

  Incompatibilité de type pour l'opérateur COUNT.

- **PCLrc = PCL44**

  Incompatibilité de type pour l'opérateur MEMBER.

- **PCLrc = PCL45**

  Incompatibilité de type pour l'opérateur INCLUDED.

- **PCLrc = PCL46**

  Incompatibilité de type pour l'opérateur ISIN.

- **PCLrc = PCL47**

  Incompatibilité de type pour l'opérateur SETOF.

- **PCLrc = PCL48**

  Incompatibilité de type pour l'opérateur WHERE.

- **PCLrc = PCL49**

  Pas de classe référencée par l'expression (no-value).

- **PCLrc = PCL50**

  Type "undetermined" pour l'expression.

- **PCLrc = PCL51**

  Expression non correct (aucun opérateur reconnu).

- **PCLrc = PCL52**

  Identifiant d'instance sous forme incorrecte (pas sous la forme "classname / nummer"), ou n'existant pas.

- **PCLrc = PCL53**

  Expression non correcte contenant l'opérateur EXIST.

- **PCLrc = PCL54**

  Expression non correcte contenant l'opérateur FORALL.

- **PCLrc = PCL55**

Essai d'évaluer une classe dite basic.

- **PCLrc = PCL56**

  Expression non correcte contenant un set explicite de valeur.

- **PCLrc = PCL57**

  Expression non correcte contenant un tuple.

Listing

Annexe 11

Vu sa taille importante (environ 80 pages), le listing reprenant le code des procédures constituant l'implémentation du langage PCL n'a pas été ajouté aux annexes. Il est cependant disponible sur demande.